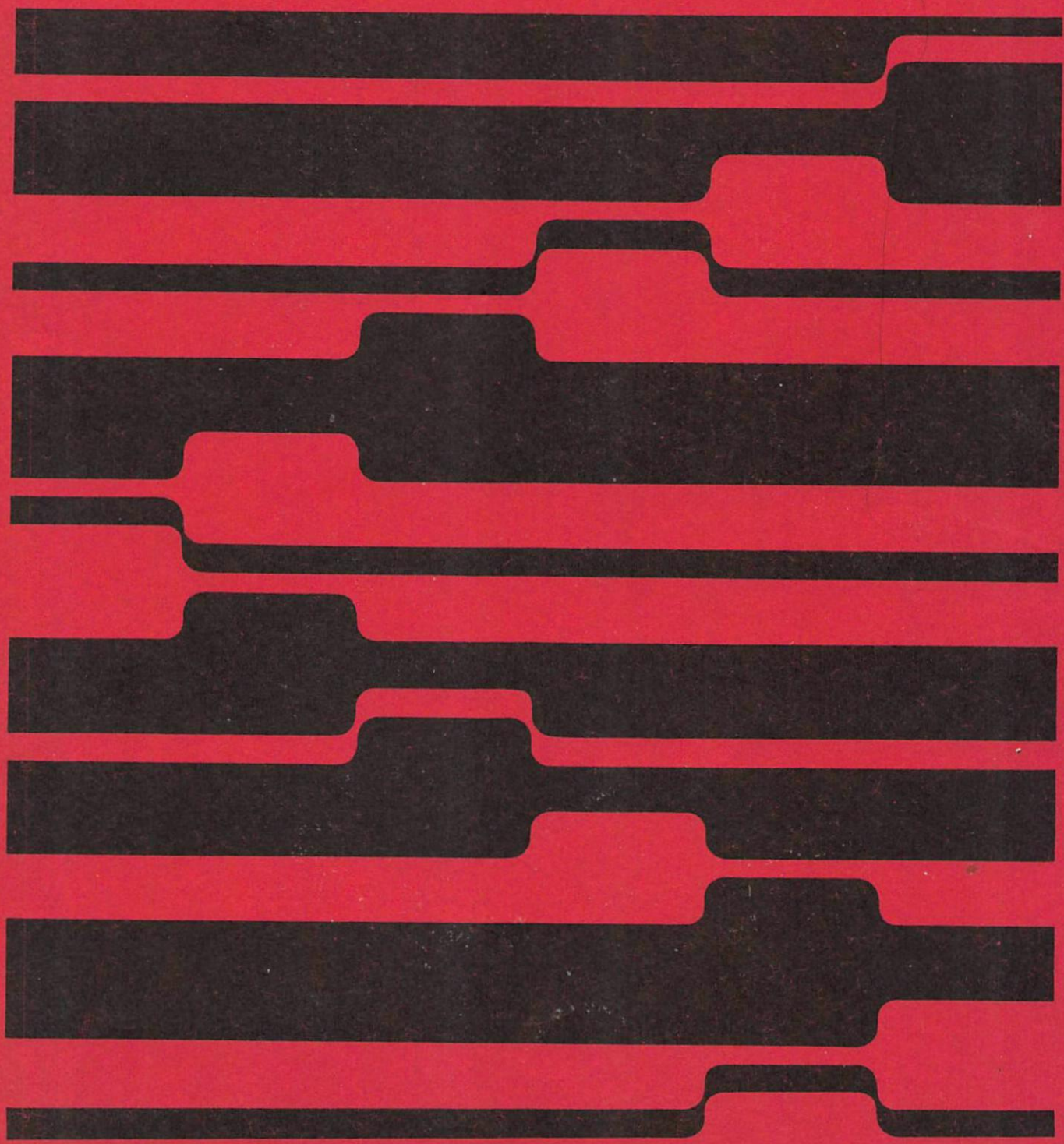
ICL

COBOL

1900 Series



ICL

COBOL

1900 Series

With effect from 9th July 1968 the name of
International Computers and Tabulators Limited
has been changed to
International Computers Limited.

Technical Publication 4082

© International Computers Limited 1968

First Edition March 1968

Reprinted November 1968
(incorporating Amendment List 1)

Reprinted March 1969
(incorporating Amendment Lists 1 and 2)

Issued by Technical Publications Service
International Computers Limited
Head Office: ICL House, Putney, London SW15
and printed in Great Britain by
ICL Printing Services, Letchworth, Hertfordshire

Preface

This manual describes the facilities available in 1900 COBOL and the operating procedures for the compiling machine. The manual is intended for reference purposes. The publication "Introduction to COBOL" is provided for those who wish to learn how to use the language.

Part 1 describes the 1900 COBOL compilers and lists the facilities available in each and the environment necessary for each. Part 2 describes the COBOL language as implemented for the 1900 series. As well as specifying the language it also includes chapters on programming techniques and the Rapidwrite features equivalent to the COBOL features described. Part 3 describes the operating procedures for running a program written in COBOL.

The following acknowledgement is reprinted from the COBOL 1965 Report issued by the U.S. Department of Defense.

Any organization interested in reproducing the COBOL report and specification in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention COBOL in acknowledgement of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

Procedures have been established for the maintenance of COBOL. Enquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference of Data Systems Languages.

The authors and copyright holders of the copyrighted material used herein

> FLOW-MATIC (Trademark for Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Note to COBOL users

This manual replaces the manuals COBOL Programming Language and COBOL User's Guide. As well as the information contained in them this manual also describes the following facilities which are not yet available. The release dates of these facilities will be announced in the usual manner.

1 The Compact COBOL compiler XE18 and the COBOL compilers XE17, XE19 and XE20 as described in Chapter 1.

2 COBOL subroutines (Linkage Section, CALL and EXIT PROGRAM) as described in Chapters 18, 5 and 6 respectively.

3 The facility of having ENTERed subroutines in overlays of the object program as described in Chapter 18.

4 The EXAMINE statement as described in Chapter 6.

5 The direct access verbs and clauses described in Chapters 4 to 6.

6 RERUN on direct access devices as described on page 32.

7 The COBOL library and COPY facilities as described in Chapters 4 to 8.

8 The facility of overlaying object programs from the upper part of core store as described on page 23.

9 The JUSTIFIED RIGHT clause as described on page 47.

Contents

Introduction	xv
CODASYL AND COBOL	xv
THE PHILOSOPHY OF COBOL	xvi
PART 1 COMPILERS	
Chapter 1 1900 COBOL compilers	1
COMPILERS	1
XE11	1
XE15	1
XE18	1
XEA3	2
XE20	2
XE1B	2
XEKA	2
XEKB	3
ENVIRONMENTAL FACILITIES	3
LANGUAGE FACILITIES	5
PART 2 PROGRAMMING IN COBOL	
Chapter 2 Structure and basic elements of a COBOL program	13
GENERAL COMMENTS	13
PROGRAM STRUCTURE	13
BASIC ELEMENTS OF THE LANGUAGE	14
Language character set	14
Basic elements used for punctuation	14
Basic elements invented by the COBOL programmer	15
LITERALS	15
Non-numeric literals	15
Numeric literals	15
USER-DEFINED NAMES	15
Data names	15
Condition names	15
Procedure names	15
Basic elements treated as reserved words by the compiler	16
KEY WORDS AND SYMBOLS	16
OPTIONAL WORDS	16
KEY WORDS WHOSE USE IS NOT CONFINED TO ONE DIVISION	16
FIGURATIVE CONSTANTS	16
KEY WORDS USED FOR QUALIFYING	17
COBOL PROGRAM SHEET	17
Sequence number	17

Continuation indicator 18
Program text 18
Identification columns 18
NOTATION 18

Chapter 3 Identification division 19

Chapter 4 Environment division 21
CONFIGURATION SECTION 22
The SOURCE-COMPUTER paragraph 22
The OBJECT-COMPUTER paragraph 23
The SPECIAL-NAMES paragraph 24
INPUT-OUTPUT SECTION 26
The FILE-CONTROL paragraph 26
The I-O-CONTROL paragraph 32

Chapter 5 Data division 33
DEFINITIONS OF TERMS 33
FILE SECTION 35
File description 36
FD FILE-NAME 36
RECORDING MODE 36
FILE CONTAINS 37
BLOCK CONTAINS 37
LABEL RECORD 37
VALUE OF ID 38
DATA RECORDS 38
SEQUENCED ON 38
Sort file description 38.1
WORKING-STORAGE SECTION 39
CONSTANT SECTION 40
LINKAGE SECTION 41
DATA DESCRIPTION SKELETON 42
Record description 42
CLASS 44
EDITING CLAUSES 45
FILLER 46
JUSTIFIED 47
OCCURS 48
PICTURE 49
POINT LOCATION 53
REDEFINES 54
RENAMES 55
SIGNED 56
SIZE 57
SYNCHRONIZED 58
USAGE 59
VALUE 62
CONDITION NAMES ASSOCIATED WITH DATA FIELDS 63
ALLOCATION OF STORAGE TO DATA 63
Synchronization 64

Chapter 6 Procedure division 65
STRUCTURE 65
Heading format 65
Procedures 65
Imperative statements 65
Compiler directing statements 66
Conditional statements 66
ELEMENTS USED IN THE PROCEDURE DIVISION 66
Data references 66
LITERALS 66
DATA NAMES 66
QUALIFIED DATA NAMES 66
SUBSCRIPTED DATA NAMES 66
Conditions 68
RELATION CONDITIONS 69
SIGN CONDITIONS 71
CLASS CONDITIONS 71
CONDITION-NAME CONDITIONS 72
SWITCH-STATUS CONDITIONS 72
COMPOUND CONDITIONS 72
STATEMENT FORMATS 73
ACCEPT 74
ADD 76
ALTER 76.1
CALL 76.2
CLOSE 77
COPY 78
DELETE 79
DISPLAY 80
DIVIDE 82
ENTER 83
EXAMINE 86
EXIT 87
GO TO 88
IF 89
MOVE 90
MULTIPLY 91
NOTE 92
OPEN 93
PERFORM 94.1
READ 97
RELEASE 98.1
RETURN 98.2
REWRITE 99
SEEK 100
SORT 100.1
STOP 101
SUBTRACT 102
WRITE 103

Chapter 7 Program control and segmentation 105

Chapter 8 The COBOL library 107

Chapter 9 Rules of arithmetic 109
INTRODUCTION 109
Definitions 109
Basic concepts 109
Types of computation 110
DECIMAL COMPUTATIONS 110
ADD and SUBTRACT 110
MULTIPLY 110
DIVIDE 110
STERLING COMPUTATIONS 111
Rounding with sterling receiving fields 112
BINARY COMPUTATIONS 112
ADD and SUBTRACT 112
MULTIPLY 112
DIVIDE 113
MIXED COMPUTATIONS 113
Numeric MOVE 113
BINARY TO DECIMAL 113
DECIMAL TO BINARY 113
ADD and SUBTRACT 114
MULTIPLY 114
DIVIDE 114

Chapter 10 Moving data in the store 117
MOVING UNEDITED ELEMENTARY FIELDS TO FIELDS OF THE SAME CLASS AND USAGE 117
1 ALPHABETIC to ALPHABETIC 117
2 ALPHANUMERIC to ALPHANUMERIC 117
3 DISPLAY NUMERIC to DISPLAY NUMERIC 117
4 DISPLAY-n to DISPLAY-n 118
5 COMPUTATIONAL to COMPUTATIONAL 118
6 COMPUTATIONAL-1 to COMPUTATIONAL-1 118
MOVING UNEDITED ELEMENTARY FIELDS TO FIELDS OF DIFFERENT CLASS OR USAGE 119
1 Receiving field ALPHABETIC 119
2 Receiving field ALPHANUMERIC 119
3 Receiving field DISPLAY NUMERIC 119
4 Receiving field DISPLAY-n 120
5 Receiving field COMPUTATIONAL 120
6 Receiving field COMPUTATIONAL-1 120
MOVING SIGNED VALUES TO UNSIGNED FIELDS AND VICE VERSA 121
MOVING SYNCHRONIZED FIELDS 121
MOVING LITERALS TO FIELDS 121
MOVING FIGURATIVE CONSTANTS TO FIELDS 122
MOVING NON-ELEMENTARY FIELDS 122

EDITING 122
1 Editing ALPHABETIC fields 122
2 Editing ALPHANUMERIC fields 123
3 Editing NUMERIC fields 123
SIMPLE INSERTION CHARACTERS 123
THE CURRENCY SYMBOL 124
REPORT SIGNS 124
ZERO SUPPRESSION 125
CHEQUE PROTECTION 126
BLANK WHEN ZERO 126

Chapter 11 Representation of data on external media 127
CHARACTER PERIPHERAL DEVICES 127
Punched cards 127
Paper tape 127
VARIABLE LENGTH FIELDS ON PAPER TAPE 128
Printing 128
MAGNETIC TAPE 128
Variable length records 128
DIRECT ACCESS DEVICES 128
VARIABLE LENGTH RECORDS IN MAGNETIC FILES 130

Chapter 12 Programming techniques 131
FORMAT DISCIPLINES 131
THE DATA DIVISION 131
Data names 131
Magnetic tape files: the record count word 132
Magnetic tape files: the block size 132
Paper tape: end marker 132
Paper tape: inter-block gaps 132
Short data blocks on paper tape or cards 133
Fillers 133
USER'S DATA ALLOCATION 133
COMPUTATIONAL fields 133
The use of FILLERS in bit fields 134
Synchronization 134
DISTRIBUTION OF INPUT DATA 135
DATA VALIDATION 136
REDEFINES 136
Possible misuse of REDEFINES 139
THE REPRESENTATION OF THE DECIMAL POINT 139
THE PROCEDURE DIVISION 140
Subscripting 140
Arithmetic: overflow and oversize operations 140
Segmentation 142
Literals 143
Notes on the verbs 143
ACCEPT FILE KEYS FROM 143
CLOSE 144
ENTER 144

PERFORM 145
STOP 147
WRITE 147

Chapter 13 Subroutines for use in COBOL programs 149
PROCESSING DIRECT ACCESS FILES BY SUBROUTINE 149
Introduction 149
Calling the subroutines 150
OPENING A FILE 150
READING A FILE 152
WRITING A RECORD 153
DELETING A RECORD 154
EXTENDING A FILE 154
CLOSING A FILE 154.1
SORTING MAGNETIC TAPE FILES BY SUBROUTINE 154.1
Input data 154.2
Output data 155
Hardware requirements 155
Calling the subroutine 156
Console typewriter messages 157
Example program using COBOLSORT 158
SORTING DIRECT ACCESS FILES BY SUBROUTINE 160
Constituent parts of the system 160
General 160.1
Keys 160.1
Calling the subroutines 160.2
NOTES ON PARAMETERS 160.2
CSORT : THE SORT ENVIRONMENT 160.2
CSORTA : SORT WITH INPUT FILE AND OUTPUT FILE 160.2
CSORTB : SORT WITH INPUT PROCEDURE AND AND OUTPUT FILE 160.3
CSORTC : SORT WITH INPUT FILE AND OUTPUT PROCEDURE 160.3
CSORTD : SORT WITH INPUT PROCEDURE AND OUTPUT PROCEDURE 160.3
CSORTE : TO RETURN TO THE COBOL STATEMENT FOLLOWING THE SORT 160.3
CPERM 160.4
CUES WITHIN INPUT AND OUTPUT PROCEDURES 160.4
Notes 160.4
Example program with input and output procedures 160.5
PROCESSING VARIABLE LENGTH FIELDS BY SUBROUTINE 160.6
Introduction 160.6
Calling the subroutine 161
Key formats 161

Error reports 164
Example programs using the subroutines 165
COBPAK 165
COBDIST 166
COBDISTM AND COBPAKM 166

Chapter 14 Rapidwrite 169
INTRODUCTION 169
The purpose of Rapidwrite 169
Using Rapidwrite 169
Input 169
Output 169
IDENTIFICATION DIVISION 170
ENVIRONMENT DIVISION 170
Configuration section 170
SOURCE COMPUTER AND OBJECT COMPUTER 170
Input-output section 170
FILE-CONTROL 170
I-O-CONTROL 170
DATA DIVISION 171
File description 171
Record description 171
PROCEDURE DIVISION 172
Linking cards 173
Rules for writing on cards 173
Procedures 174
ACCEPT 174
ADD 174
ALTER 174
CLOSE 175
COMPUTE 175
DISPLAY 175
DIVIDE 176
ENTER 176
EXIT 176
GO 176
IF 176
INCLUDE 177
MOVE 177
MULTIPLY 177
NOTE 177
OPEN 177
PARAGRAPH 177
PERFORM 178
READ 178
SECTION 178
STOP 178
SUBSCRIPTS 179
SUBTRACT 179
WRITE 179

PART 3 OPERATING PROCEDURES

Chapter 15 The source program 181
FORMAT OF THE SOURCE PROGRAM 181
On 80-column cards 181
On paper tape 182
On magnetic tape 183
Format of a Rapidwrite source program 183
PREPARATION OF THE SOURCE PROGRAM 183
Segmentation 183
Check list 184
STEERING LINES 185
Use of the steering lines 186
Format of a fast peripheral operand unit 187
List of directives 188
ORDER OF INPUT 191
COMPILER OPERATING INSTRUCTIONS 191
General 191
Loading 191
Switch settings 192
Entry points 192
End messages 192
Special action during consolidation 193
Files names 193
Compiler messages 193
Messages output by the general purpose loader 193
Messages output by the overlay packages 194
Sample operating instruction sheet 195

Chapter 16 Compiler listings and other printouts 197
INITIAL MESSAGES 197
THE SOURCE PROGRAM 197
THE DATA MAPS 197
THE PROGRAM MAPS AND SEGMENT LISTING 198
The consolidator listing 199
ERROR MESSAGES 199
Format and parameter errors 201
COPY phase errors 204.1
COBOL language errors 204.2
Code list 234.3
OTHER COMPILE TIME MESSAGES 238
TERMINAL MESSAGES OF COMPILATION 239
Printer 239
Console typewriter 240

Chapter 17 The object program 241
FORMAT 241
ALLOCATION OF STORAGE IN THE OBJECT 241
The *LOWER steering line 242
PERIPHERAL HANDLING IN THE OBJECT PROGRAM 243
Logic of the instructions generated for peripheral handling 244

Labels and sentinels 246
Dumping and restarts (rerun) 246
File numbering in the object program 247
OPERATING INSTRUCTIONS FOR THE OBJECT PROGRAM 247
Running a consolidated object program 247
Run-time error messages and halts 247
The *LOAD steering line 251
ABORTIVE OBJECT RUNS 251
RUN-TIME SUBROUTINES 251
SORTING IN THE OBJECT PROGRAM 254.1

Chapter 18 Own code, library and COBOL subroutines 255
COBOL SUBROUTINES 255
Example of a COBOL subroutine 255
CONSOLIDATION OF COBOL PROGRAMS AND INCLUSION OF COBOL OR OTHER LANGUAGE SUBROUTINES 256
Consolidation as part of the COBOL compiler 257
DEFINITIONS 257
FORMATS 258
THE "FORCE-IN" BIT 258
THE PAUSE BLOCK 259
ORDER OF SUBROUTINES 259
Independent consolidation 259

Chapter 19 Testing and correcting 261
MONITORING THE SOURCE PROGRAM 261
CORRECTING THE SOURCE PROGRAM 262
Deletions 262
Insertions 263
Replacements 263

Chapter 20 Compiler limits 265

PART 4 APPENDIXES
Appendix 1 Alphanumeric data character set 271
Appendix 2 Reserved words 273
Appendix 3 Facsimiles of Rapidwrite forms and cards 275
Appendix 4 1900 COBOL mass storage verbs 279
Appendix 5 1900 Series COBOL example 281
Appendix 6 General description of the compilers 289

Index 293

Introduction

CODASYL AND COBOL

CODASYL (Conference on Data Systems Languages) and COBOL (Common Business Oriented Language) were conceived at a meeting held in 1959. At that time the following general objective was stated:

> "The current experience of users of electronic data processing equipment indicates that a major problem in the efficient utilization of such equipment lies in the inability to state the data processing application in such a way that computer programs are developed and maintained with a minimum of time and programming effort.
>
> "COmmon Business Oriented Language, independent of any make or model of computer, open ended, and stated in both an English notation and a narrative form, would do much to solve or to ameliorate this problem. Such a language would also simplify and speed up the solution of the related problem of training personnel in the design of data processing systems and in the development of computer programs for such systems.
>
> "In general, the development of a common language would serve to benefit the user in the following situations.
>
> "In developing data processing systems for existing computers, it is important that these systems be capable of processing on future, more powerful computers of any manufacturer with a minimum of conversion costs.
>
> "Full documentation of present systems in a form conducive to making changes and additions with minimum expenditures of time and cost is necessary in order to meet effectively the rapidly changing and expanding requirements of management.
>
> "The need to produce a large number of computer programs in a short period of time often requires the augmentation of programming staffs by the addition of relatively inexperienced programmers."

On May 28 and 29, 1959 a meeting was held in the Pentagon for the purpose of considering the desirability and feasibility of establishing a common programming language for commercial applications. Representatives from users, both in private industry and in the United States government, computer manufacturers and other interested parties were present.

The result of the meeting was an agreement to go ahead with the project and to establish three committees, short range, intermediate range and long range to pursue the matter. The short range committee was given the task of developing an immediate language and was instructed to take the best of three existing language-compiler systems, FLOWMATIC, AIMACO and Commercial Translator and to produce a language superior to any of these.

By September 1959 the short range committee had specified a language which they considered superior to existing language compiler systems. This language specification was further modified and by December 1959 COBOL existed as a language which was not identified with any manufacturer and presented advantages for both government and private industry users. COBOL as it then existed was specified in a report published by the United States Government Printing Office and has since become known as COBOL-60.

The second official version of COBOL, COBOL-61, was published in 1961. It was not completely compatible with COBOL-60 but it is the version which was generally implemented and it forms the basis for most COBOL compilers in use today. Other versions of COBOL since then have consisted basically of COBOL-61 with various additional facilities and modifications.

The effort to define a COBOL standard began in January 1963 and currently involves the United States of America Standards Institute (USASI), the European Computer Manufacturers Association (ECMA) and the International Organization for Standards (ISO). The USASI's chief contribution to this has been to issue the COBOL Information Bulletin at regular intervals giving information on the development of COBOL.

THE PHILOSOPHY OF COBOL

The task of specifying COBOL was undertaken because of the computer user's need for a problem oriented machine-independent language for commercial applications. To be successful such a language should have certain characteristics. Programs written in the language should be capable of easy conversion from one machine to another, the language should be easily understandable in format and notation, it should provide good program documentation and it should be capable of specifying data processing problems in such a way that an implementation of the language by means of a compiler can produce efficient object code.

COBOL is a language for programmers. It is not intended to be used by people unfamiliar with computers. The attributes of the language make it more feasible for use by applications oriented personnel and make it easier to teach to novices, but an understanding of programming and data processing concepts is a pre-requisite to the writing of efficient COBOL programs.

The principal benefits of COBOL are the ability to write programs quickly and efficiently. Other major benefits which improve the overall efficiency of the data processing function accrue through the use of COBOL, especially in the fields of documentation, understandability and compatibility.

The completeness of program documentation produced by a COBOL compilation is dependent upon the particular compiler and upon the amount of thought and effort expended by the programmer. Because of the narrative nature and English-like syntax of the language, however, a certain amount of documentation is inherent. If the user establishes programmer guidelines so that data-names and procedure-names are consistent and meaningful, the listings produced by a compilation will constitute a good basis for documentation. Generally, the quality and quantity of documentation produced by the use of COBOL is superior to that of other languages. This superiority is very important for the maintenance and revision of production programs.

The organization of the language, both as to data hierarchies and procedure statements, results in COBOL being much easier to teach and understand than machine oriented languages or assembly systems. This attribute makes COBOL very useful for man-to-man as well as man-to-machine communications. However, it is necessary to keep in mind the importance of comprehensive training, including basic computer logic and programming fundamentals as well as COBOL training itself.

While COBOL is not a completely machine-independent language, it closely approaches this goal. The user must exercise some degree of care if he wishes to produce highly compatible programs. It is appropriate here to examine, division by division, the degree of machine-independence in a COBOL program.

The Identification Division is totally machine-independent. However, since it does not produce object code, this machine-independence is not of great consequence to the user.

The Environment Division is completely machine-dependent. Those aspects of the data processing problem that are related to a particular hardware system are specified here. When a COBOL program is converted from one computer to another this division must be at least partially rewritten. The Environment Division usually consists of only three or four paragraphs so that the effort of rewriting it is not great.

The Data Division is compatible across machine lines in so far as the data being described are compatible. For example, changes will be required in the Data Division if the user is converting from a character, variable word length machine to a fixed word length machine with more than one internal representation.

The Procedure Division is compatible across machine lines to the extent that the language can separate the problem from the hardware. If, for example, the collating sequence of a new machine is different from that of the machine currently in use, certain statements in the Procedure Division will have to be rewritten. Apart from this "language independent" circumstance, the Procedure Division should convert from one machine to another without any change.

PART 1 I.C.T. COBOL COMPILERS

Chapter 1 1900 COBOL compilers

There are two versions of 1900 COBOL, namely Compact COBOL and COBOL, for each of which there are various compilers available. The compiler that is used in any particular instance depends upon the size of the central processor, the configuration available and the media being used as well as the version of COBOL concerned.

A brief description of each of the compilers is given below.

Rapidwrite input is prepared for the COBOL compilers by means of the free-standing Rapidwrite translator XE88, which will be described in a User Notice.

COMPILERS

Compact COBOL compiler XE11

XE11 is a Compact COBOL Compiler on magnetic tape for machines of 8K and above words of store. Source programs are input from paper tape, punched cards or magnetic tape and semi-compiled object programs may be output to paper tape, punched cards or magnetic tape. The compiler will produce object programs that can access data files held on punched cards, paper tape, magnetic tape or exchangeable discs (in the latter case by means of ENTERed subroutines). Excluding Executive, XE11 uses 5888 words of core store.

The necessary environment for XE11 is as follows:

One central processor with not less than 5888 Executive-free words.

One paper tape reader or one card reader.

One line printer.

Four magnetic tape decks.

One console typewriter.

One card or paper tape punch if the object program is required on cards or paper tape.

Compact COBOL compiler XE15

XE15 is a Compact compiler on cassette tape for use on central processors with or without a console typewriter. Source programs are input from paper tape or punched cards and the semi-compiled object programs may be output to paper tape, punched cards or cassette tape. The compiler will produce object programs that can access data files held on punched cards, paper tape, cassette tape or magnetic tape. Excluding Executive, XE15 uses 5888 words of core store.

The necessary environment for XE15 is as follows:

One central processor with not less than 5888 Executive-free words.

On paper tape reader or one card reader.

Four cassette tape stations.

One line printer.

One card or paper tape punch if the object program is required on cards or paper tape.

One console typewriter (optional).

Compact COBOL compiler XE18

XE18 is a Compact COBOL compiler on exchangeable discs for machines of 8K and above words of store. Source programs are input from paper tape, punched cards or exchangeable discs and the semi-compiled object programs may be output to exchangeable discs, paper tape or punched cards. The compiler will produce object programs that can access data files held on punched cards, paper tape, magnetic tape or exchangeable discs (in the latter case by means of ENTERed subroutines). Excluding Executive, XE18 uses 5888 words of core store.

The necessary environment for XE18 is as follows:

One central processor with not less than 5888 Executive-free words.

One paper tape reader or one card reader.

One line printer.

Two exchangeable disc transports.

One console typewriter.

One card or paper tape punch if the object program is required on cards or paper tape.

COBOL compiler XEA3

XEA3 is a COBOL batch compiler on magnetic tape for machines of 16K and above words of store. Source programs are input from paper tape, punched cards or magnetic tape and semi-compiled object programs may be output to paper tape, punched cards or magnetic tape. The compiler will produce object programs that can access files held on punched cards, paper tape, magnetic tape and direct access devices (in the latter case by means of ENTERed subroutines). Excluding Executive, the compiler uses 10240 words of core store.

The necessary environment for XEA3 is as follows:

One central processor with not less than 10240 Executive-free words.

One paper tape reader or one card reader.

One line printer.

Four magnetic tape decks.

One console typewriter.

One card or paper tape punch if the object program is required on cards or paper tape.

COBOL compiler XE20

XE20 is a COBOL compiler on exchangeable discs for machines of 16K and above words of store. Source programs are input from paper tape, punched cards, magnetic tape or exchangeable discs and semi-compiled object programs may be output to paper tape, punched cards, magnetic tape or exchangeable discs. The compiler will produce object programs that can access files held on punched cards, paper tape, magnetic tape and exchangeable and fixed discs (in the latter case by means of the COBOL Direct Access verbs). Excluding Executive, XE20 uses 12032 words of core store.

The necessary environment for XE20 is as follows:

One central processor with not less than 12032 Executive-free words.

One paper tape reader or one card reader.

One line printer.

Two E.D.S. or Twin-E.D.S. transports, or four magnetic tape decks.

One console typewriter.

One card or paper tape punch if the object program is required on cards or paper tape.

COBOL compiler XE1B

The environmental facilities of XE1B are the same as those of XE20, with additional limited facilities for handling magnetic tape subfiles.

The language facilities are the same as those of XEA3, but with the addition of the facility to compile COBOL subroutines.

COBOL compiler XEKA

XEKA is a COBOL batch compiler on magnetic tape or exchangeable discs for machines of 16K and above words of core store. The compiler will operate in any amount of core store between 12K and 31K and will take advantage of large store allocations to compile larger source programs and/or to compile programs faster by optimising

its overlay structure. Source programs may be input from, and semi-compiled object programs may be output to, paper tape, punched cards, magnetic tape or exchangeable discs. Source programs may be compiled as COBOL subroutines and consolidated together later. Source programs may also be corrected and compiled in a batch, and entries from a COBOL library may be incorporated at compile-time. The compiler will produce object programs that can access files held on paper tape, cards, magnetic tape and discs.

The minimum configuration for XEKA is as follows:

One central processor with not less than 12032 Executive-free words.

One paper tape or card reader.

One line printer.

Four magnetic tape decks or

Two E.D.S. transports or two Twin-E.D.S. transports, or a combination of magnetic tape and discs, but not E.D.S. and Twin-E.D.S.

One console typewriter.

One card or paper tape punch if the object program is required on cards or paper tape.

COBOL compiler XEKB

As for XEKA except that the overlay optimisation is not available i.e. only 16K overlays are present.

ENVIRONMENTAL FACILITIES

The following tables summarize the various 1900 COBOL compilers. Table 1 summarizes the environmental description of the compilers and Table 2 details the language facilities available with each compiler.

Users should note that all the compilers listed in Table 1 can use the Sort and Variable-length Subroutines described in Chapter 13. However, in the case of the Sort subroutines, users with only 8K words of store should note the storage requirements for the subroutine on page 113.

Users should also note that only those compilers described in Table 1 as being able to access data on a particular direct access device can use the direct access subroutines described in Chapter 13 to access data on the device specified in the table.

Compiler	Version of COBOL	Compiler Medium	Minimum Environment												Accessing Data On						
			Machine Core Size	Paper Tape Reader	Paper Tape Punch	Card Reader	Card Punch	Line Printer	Console Typewriter	Magnetic Tape	Cassette Tape	Exchangeable Discs	Fixed Discs		Paper Tape	Cards	Magnetic Tape	Cassette Tape	Exchangeable Discs	Fixed Discs	
XE11	COMPACT	MT	8K	A	O	A	O	*	*	4	-	-	-		X	X	X	-	X	X	
XE15	COMPACT	CT	8K	A	O	A	O	*	O	-	4	-	-		X	X	X	X	-	-	
XE18	COMPACT	E.D.S.	8K	A	O	A	O	*	*	-	-	2	-		X	X	X	-	X	X	
XE20	COBOL	MT/E.D.S.	16K	A	O	A	O	*	*	4A	-	2A	-		X	X	X	-	X	X	
XE1B	COBOL	MT/E.D.S.	16K	A	O	A	O	*	*	4A	-	2A	-		X	X	X	-	X	X	
XEA3	COBOL	MT	16K	A	O	A	O	*	*	4	-	-	-		X	X	X	-	X	X	
XEKA/ XEKB	COBOL	MT/E.D.S.	16K	A	O	A	O	*	*	4A	-	2A	-		X	X	X	-	X	X	

* = Required - = Not required
A = Alternative
O = Optional

X = Accessible medium
- = Inaccessible medium

Table 1 The 1900 COBOL compilers

LANGUAGE FACILITIES

Table 2 Facilities available with 1900 COBOL compilers

LANGUAGE CONCEPTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
Character Set							
For names : 0-9, A-Z, -	✓	✓	✓	✓	✓	✓	✓
For punctuation: [.], ["], [(], [)], space, comma, semicolon	✓	✓	✓	✓	✓	✓	✓
For relations: [=] [>] [<]	✓	✓	✓	✓	✓	✓	✓
For editing: [B], [0], [+], [-], [.], [,], [CR], [DB], [Z], [*], [£], [$]	✓	✓	✓	✓	✓	✓	✓
Name characteristics				✓	✓	✓	✓
Maximum size is 30 characters	✓	✓	✓	✓	✓	✓	✓
Data Name							
Must contain an alpha character	✓	✓	✓	✓	✓	✓	✓
Condition Name				✓	✓	✓	✓
Procedure Name							
Name may be all numeric	✓	✓	✓	✓	✓	✓	✓
Literals: 1 to 120 characters	✓	✓	✓	✓	✓	✓	✓
Figurative Constants							
ZERO				✓	✓	✓	✓
ZEROS	✓	✓	✓	✓	✓	✓	✓
ZEROES				✓	✓	✓	✓
SPACE				✓	✓	✓	✓
SPACES	✓	✓	✓	✓	✓	✓	✓
HIGH-VALUE				✓	✓	✓	✓
HIGH-VALUE				✓	✓	✓	✓
LOW-VALUE				✓	✓	✓	✓
LOW-VALUES				✓	✓	✓	✓
QUOTE				✓	✓	✓	✓
QUOTES				✓	✓	✓	✓
ALL				✓	✓	✓	✓
Reserved Words							
Key words	✓	✓	✓	✓	✓	✓	✓
Optional Words	✓	✓	✓	✓	✓	✓	✓
Connectors							
Qualifier OF IN				✓	✓	✓	✓
Series,(comma)	✓	✓	✓	✓	✓	✓	✓
Logical AND OR				✓	✓	✓	✓
Qualification				✓	✓	✓	✓
Subscripting							
3 levels	✓	✓	✓	✓	✓	✓	✓

IDENTIFICATION DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
PROGRAM-ID clause	✓	✓	✓	✓	✓	✓	✓
AUTHOR clause	✓	✓	✓	✓	✓	✓	✓
INSTALLATION clause	✓	✓	✓	✓	✓	✓	✓
DATE-WRITTEN clause	✓	✓	✓	✓	✓	✓	✓
DATE-COMPILED clause	✓	✓	✓	✓	✓	✓	✓
SECURITY clause	✓	✓	✓	✓	✓	✓	✓
REMARKS clause	✓	✓	✓	✓	✓	✓	✓
ENVIRONMENT DIVISION ELEMENTS							
Configuration section							
SOURCE-COMPUTER paragraph	✓	✓	✓	✓	✓	✓	✓
COPY option						✓	✓
COPY REPLACING option						✓	✓
Computer-name	✓	✓	✓	✓	✓	✓	✓
OBJECT-COMPUTER paragraph	✓	✓	✓	✓	✓	✓	✓
COPY option						✓	✓
COPY REPLACING option						✓	✓
Computer-name	✓	✓	✓	✓	✓	✓	✓
MEMORY SIZE clause	✓	✓	✓	✓	✓	✓	✓
HAND-KEYS clause		✓					
SEGMENT-LIMIT clause	✓	✓	✓	✓	✓	✓	✓
ASSIGN OBJECT-PROGRAM clause	✓	✓	✓	✓	✓	✓	✓
SPECIAL-NAMES paragraph				✓	✓	✓	✓
COPY option						✓	✓
COPY REPLACING option						✓	✓
hardware-name clause				✓	✓	✓	✓
TYPE clause				✓	✓	✓	✓
mnemonic-name clause				✓	✓	✓	✓
ON STATUS clause				✓	✓	✓	✓
OFF STATUS clause				✓	✓	✓	✓
DECIMAL POINT							✓
*DATE				✓	✓	✓	✓
*TIME				✓	✓	✓	✓
Input-output section							
FILE-CONTROL paragraph	✓	✓	✓	✓	✓	✓	✓
COPY option						✓	✓
COPY REPLACING option						✓	✓
SELECT...ASSIGN clause	✓	✓	✓	✓	✓	✓	✓
MULTIPLE REEL clause	✓	✓	✓	✓	✓	✓	✓

ENVIRONMENT DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
RESERVE ALTERNATE AREA(S) clause	✓	✓	✓	✓	✓	✓	✓
ACCESS MODE clause				✓			✓
SEQUENTIAL				✓			✓
RANDOM				✓			✓
PROCESSING MODE clause				✓			✓
SEQUENTIAL				✓			✓
ORGANIZATION clause				✓			✓
INDEXED				✓			✓
DIRECT				✓			✓
FIRST-LEVEL							✓
SECOND-LEVEL							✓
ACTUAL KEY clause				✓			✓
SYMBOLIC KEY clause				✓			✓
SELECT Series	✓	✓	✓	✓	✓	✓	✓
I-O-CONTROL Paragraph	✓	✓	✓	✓	✓	✓	✓
COPY option						✓	✓
COPY REPLACING option						✓	✓
APPLY...TO TRANSFER-REPLY clause				✓			✓
APPLY...TO INDEX-BUFFER clause				✓			✓
APPLY ... TO SORT-* clause							✓
APPLY SENTINEL-PROCESSING clause				✓			✓
APPLY FILE-MESSAGES clause				✓			✓
RERUN clause	✓	✓	✓	✓	✓	✓	✓
SAME [RECORD] AREA clause				✓	✓	✓	✓
DATA DIVISION ELEMENTS							
Section headers							
FILE SECTION	✓	✓	✓	✓	✓	✓	✓
WORKING-STORAGE SECTION	✓	✓	✓	✓	✓	✓	✓
CONSTANT SECTION				✓	✓	✓	✓
LINKAGE SECTION					✓		✓
File description clauses							
COPY option						✓	✓
COPY REPLACING option						✓	✓
RECORDING MODE 2				✓	✓	✓	✓
F or V							✓
FILE CONTAINS				✓	✓	✓	✓

DATA DIVISION ELEMENTS	Compact COBOL compilers			COBOL compilers			
	XE11	XE15	XE18	XE20	XE1B	XEA3	XEKA XEKB
BLOCK CONTAINS	✓	✓	✓	✓	✓	✓	✓
RECORD CONTAINS				✓	✓	✓	✓
LABEL RECORD(S)	✓	✓	✓	✓	✓	✓	✓
STANDARD option	✓	✓	✓	✓	✓	✓	✓
OMITTED option	✓	✓	✓	✓	✓	✓	✓
GENERATION-NO option	✓	✓	✓	✓	✓	✓	✓
VALUE OF ID							
literal	✓	✓	✓	✓	✓	✓	✓
dataname				✓			✓
ACTIVE-TIME							
literal	✓	✓	✓	✓	✓	✓	✓
dataname				✓			✓
GENERATION-NO dataname				✓			✓
DATA RECORDS	✓	✓	✓	✓	✓	✓	✓
SEQUENCED ON				✓	✓	✓	✓
Sort file description clause							
BLOCK CONTAINS							✓
Record description clauses							
COPY option						✓	✓
COPY REPLACING option						✓	✓
Level Numbers 1-10	✓	✓	✓	✓	✓	✓	✓
1-49				✓	✓	✓	✓
66				✓	✓	✓	✓
77	✓	✓	✓	✓	✓	✓	✓
88				✓	✓	✓	✓
CLASS				✓	✓	✓	✓
Editing clauses							
ZERO SUPPRESS				✓	✓	✓	✓
CHECK PROTECT				✓	✓	✓	✓
FLOAT CURRENCY SIGN				✓	✓	✓	✓
LEAVING integer PLACES				✓	✓	✓	✓
BLANK WHEN ZERO	✓	✓	✓	✓	✓	✓	✓
FILLER	✓	✓	✓	✓	✓	✓	✓
JUSTIFIED RIGHT				✓	✓	✓	✓
OCCURS	✓	✓	✓	✓	✓	✓	✓
OCCURS DEPENDING							✓

DATA DIVISION ELEMENTS	Compact COBOL compilers			COBOL compilers			
	XE11	XE15	XE18	XE20	XE1B	XEA3	XÉKA XEKB
PICTURE	√	√	√	√	√	√	√
Data Character A	√	√	√	√	√	√	√
X	√	√	√	√	√	√	√
9	√	√	√	√	√	√	√
1	√	√	√	√	√	√	√
Operational Symbols S	√	√	√	√	√	√	√
V	√	√	√	√	√	√	√
P	√	√	√	√	√	√	√

DATA DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
Fixed Insertion Characters 0	✓	✓	✓	✓	✓	✓	✓
,	✓	✓	✓	✓	✓	✓	✓
.	✓	✓	✓	✓	✓	✓	✓
B	✓	✓	✓	✓	✓	✓	✓
£	✓	✓	✓	✓	✓	✓	✓
$				✓	✓	✓	✓
+, -	✓	✓	✓	✓	✓	✓	✓
DB,CR	✓	✓	✓	✓	✓	✓	✓
Floating Characters Z	✓	✓	✓	✓	✓	✓	✓
*	✓	✓	✓	✓	✓	✓	✓
£	✓	✓	✓	✓	✓	✓	✓
$				✓	✓	✓	✓
+, -	✓	✓	✓	✓	✓	✓	✓
POINT LOCATION				✓	✓	✓	✓
REDEFINES	✓	✓	✓	✓	✓	✓	✓
RENAMES				✓	✓	✓	✓
SIGNED				✓	✓	✓	✓
SIZE				✓	✓	✓	✓
SYNCHRONIZED	✓	✓	✓	✓	✓	✓	✓
USAGE	✓	✓	✓	✓	✓	✓	✓
VALUE	✓	✓	✓	✓	✓	✓	✓
PROCEDURE DIVISION ELEMENTS							
Conditions							
Logical operators				✓	✓	✓	✓
Relation Condition	✓	✓	✓	✓	✓	✓	✓
NOT option	✓	✓	✓	✓	✓	✓	✓
Abbreviated Relation Conditions							✓
Sign Condition				✓	✓	✓	✓
NOT option				✓	✓	✓	✓
Class Condition				✓	✓	✓	✓
NOT option				✓	✓	✓	✓
Condition-name Condition				✓	✓	✓	✓
NOT option				✓	✓	✓	✓
Switch Status Condition				✓	✓	✓	✓
NOT option				✓	✓	✓	✓
Comparison							
Numeric operands	✓	✓	✓	✓	✓	✓	✓
Non-numeric operands	✓	✓	✓	✓	✓	✓	✓

PROCEDURE DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
Arithmetic operands limited to 12 digits	√	√	√	√	√	√	√
Segmentation							
Priority Numbers	√	√	√	√	√	√	√
Fixed memory 0-49	√	√	√	√	√	√	√
Non-fixed memory 50-99	√	√	√	√	√	√	√
SEGMENT-LIMIT clause	√	√	√	√	√	√	√
ACCEPT	√	√	√	√	√	√	√
FROM option				√	√	√	√
FILE KEYS FROM option				√	√	√	√
ADD	√	√	√	√	√	√	√
Series option				√	√	√	√
TO option	√	√	√	√	√	√	√
GIVING option	√	√	√	√	√	√	√
ROUNDED option	√	√	√	√	√	√	√
SIZE ERROR option	√	√	√	√	√	√	√
CORRESPONDING							√
ALTER	√	√	√	√	√	√	√
series option	√	√	√	√	√	√	√
CALL					√		√
CLOSE	√	√	√	√	√	√	√
series option	√	√	√	√	√	√	√
REEL option				√	√	√	√
NO REWIND option				√	√	√	√
LOCK option				√	√	√	√
COPY						√	√
REPLACING option						√	√
DELETE				√			√
INVALID KEY option				√			√
DISPLAY	√	√	√	√	√	√	√
UPON option				√	√	√	√
series option	√	√	√	√	√	√	√
DIVIDE	√	√	√	√	√	√	√
GIVING option	√	√	√	√	√	√	√
ROUNDED option	√	√	√	√	√	√	√
SIZE ERROR option	√	√	√	√	√	√	√
ENTER	√	√	√	√	√	√	√

PROCEDURE DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
USING option	✓	✓	✓	✓	✓	✓	✓
EXAMINE				✓		✓	✓
TALLYING option				✓		✓	✓
UNTIL FIRST option				✓		✓	✓
ALL option				✓		✓	✓
LEADING option				✓		✓	✓
REPLACING BY option				✓		✓	✓
REPLACING option				✓		✓	✓
ALL option				✓		✓	✓
LEADING option				✓		✓	✓
FIRST option				✓		✓	✓
UNTIL FIRST option				✓		✓	✓
EXIT	✓	✓	✓	✓	✓	✓	✓
PROGRAM option					✓		✓
GO TO	✓	✓	✓	✓	✓	✓	✓
DEPENDING ON option				✓	✓	✓	✓
IF	✓	✓	✓	✓	✓	✓	✓
OTHERWISE option				✓	✓	✓	✓
ELSE option				✓	✓	✓	✓
MOVE	✓	✓	✓	✓	✓	✓	✓
series option				✓	✓	✓	✓
CORRESPONDING							✓
MULTIPLY	✓	✓	✓	✓	✓	✓	✓
GIVING option	✓	✓	✓	✓	✓	✓	✓
ROUNDED option	✓	✓	✓	✓	✓	✓	✓
SIZE ERROR option	✓	✓	✓	✓	✓	✓	✓
NOTE	✓	✓	✓	✓	✓	✓	✓
sentence	✓	✓	✓	✓	✓	✓	✓
paragraph	✓	✓	✓	✓	✓	✓	✓
OPEN	✓	✓	✓	✓	✓	✓	✓
INPUT option	✓	✓	✓	✓	✓	✓	✓
series option	✓	✓	✓	✓	✓	✓	✓
OUTPUT option	✓	✓	✓	✓	✓	✓	✓
series option	✓	✓	✓	✓	✓	✓	✓
INPUT-OUTPUT option				✓			✓
series option				✓			✓

PROCEDURE DIVISION ELEMENTS	Compact COBOL compilers XE11	XE15	XE18	COBOL compilers XE20	XE1B	XEA3	XEKA XEKB
PERFORM	✓	✓	✓	✓	✓	✓	✓
THRU option	✓	✓	✓	✓	✓	✓	✓
TIMES option				✓	✓	✓	✓
UNTIL option				✓	✓	✓	✓
VARYING option				✓	✓	✓	✓
UNTIL . . . AFTER option				✓	✓	✓	✓
READ	✓	✓	✓	✓	✓	✓	✓
INTO option				✓	✓	✓	✓
AT END option	✓	✓	✓	✓	✓	✓	✓
INVALID KEY option				✓			✓
RELEASE							✓
FROM option							✓
RETURN							✓
INTO option							✓
AT END option							✓
REWRITE				✓			✓
FROM option				✓			✓
SEEK				✓			✓
SORT (all options)							✓
STOP	✓	✓	✓	✓	✓	✓	✓
literal option	✓	✓	✓	✓	✓	✓	✓
RUN option	✓	✓	✓	✓	✓	✓	✓
SUBTRACT	✓	✓	✓	✓	✓	✓	✓
series option	✓	✓	✓	✓	✓	✓	✓
GIVING option	✓	✓	✓	✓	✓	✓	✓
ROUNDED option	✓	✓	✓	✓	✓	✓	✓
SIZE ERROR option	✓	✓	✓	✓	✓	✓	✓
CORRESPONDING							✓
WRITE	✓	✓	✓	✓	✓	✓	✓
FROM option				✓	✓	✓	✓
BEFORE option				✓	✓	✓	✓
AFTER option	✓	✓	✓	✓	✓	✓	✓
ADVANCING option	✓	✓	✓	✓	✓	✓	✓
Identifier LINES option	✓	✓	✓	✓	✓	✓	✓
integer LINES option	✓	✓	✓	✓	✓	✓	✓
hardware name option	✓	✓	✓	✓	✓	✓	✓
mnemonic name option				✓	✓	✓	✓
INVALID key option				✓			✓

PART 2 PROGRAMMING IN COBOL

Chapter 2 Structure and basic elements of a COBOL program

GENERAL COMMENTS

A program written in COBOL is known as a source program. This source program is translated by means of a program called the COBOL compiler into an object program in machine language. The object program so prepared will run on a specified computer and produce answers to a given data processing problem. A source program will contain the following information:

1 An identification of the program.

2 A description of the equipment to be used to compile the source program and run the object program.

3 A description of the data to be processed.

4 A set of procedures to specify the operations to be performed on the data.

COBOL is divided into four divisions, one for each of the above items.

It is also possible to write subroutines in COBOL. These have the same structure as COBOL source programs and can be translated by some compilers into subroutines in semi-compiled form. See chapter 18 for details.

PROGRAM STRUCTURE

The text of a program consists of the four divisions in the order mentioned above. Each division starts with its heading beginning in column 8 of the Program Sheet. The headings are:

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

DATA DIVISION.

PROCEDURE DIVISION.

There must be one or more spaces between the division name and the word DIVISION and there must be a full stop after the word DIVISION. The full stop may follow the word immediately or one or more spaces may intervene.

Each division is divided either into sections (which are further divided into paragraphs) or into paragraphs.

A section starts with its heading beginning in column 8 as follows:

section-name SECTION [integer].

The section name is either invented by the programmer or is a fixed name, depending upon the division. There must be one or more spaces between the section name and the word SECTION. The word SECTION is either followed by a full stop or, in the Procedure Division, sometimes by an integer followed by a full stop. There need be no spaces between the full stop and the word preceding it (although they are allowed) but there must be at least one space between the word SECTION and the integer. For the meaning of the integer see Chapter 7.

A paragraph begins with its heading which starts in column 8 and consists of the paragraph name followed immediately or after intervening spaces by a full stop. The paragraph name is either invented by the programmer or is a fixed name, depending upon the division.

Paragraphs are divided into one or more sentences. The end of a sentence is marked by a full stop. Its beginning is also marked by a full stop, either the one which marks the end of a previous sentence or the one which marks the end of a heading. A full stop is the character ⊡ followed by at least one space.

BASIC ELEMENTS OF THE LANGUAGE

Each sentence consists of one or more of the basic elements of the language arranged so as to be meaningful to the compiler.

The compiler is provided with rules which enable it to determine where one basic element ends and the next begins. The most general rule for the COBOL programmer to remember is that one or more spaces can always be used to separate two basic elements. However, some basic elements need not be separated from one or both of their neighbouring elements by spaces. These will be described individually.

Language character set

The character set from which the basic elements are composed is

0 to 9

A to Z

Space

\+ Plus sign

\- Minus sign or hyphen

* Asterisk

/ Oblique stroke

= Equal sign

£ Pound sign

$ Dollar sign

. Full stop or decimal point

, Comma

; Semicolon

" Quotation mark

(Left parenthesis

) Right parenthesis

> "Greater than" symbol

< "Less than" symbol

Non-numeric literals form an exception to the above. A non-numeric literal can contain any of the characters which may appear in an alphanumeric field when running the object program, with the exception of the quotation mark. This character set is given in Appendix 1.

Basic elements used for punctuation

A *full stop* is used to end a sentence or a heading. It is the symbol [.] . It must be followed by at least one space but need not be immediately preceded by a space.

A *semi-colon* (symbol [;]) and a comma (symbol [,]) may be written between any two basic elements in the text. They must be followed by at least one space. Both are ignored by the compiler.

A *left parenthesis,* that is the symbol [(] , is usually classified as a punctuation symbol. Its use will be described where appropriate later in the manual. It need not be followed or preceded by a space when used as a punctuation mark (i. e. in all cases except when used in a Picture).

A *right parenthesis,* that is the symbol [)] , is also usually classified as a punctuation symbol. Its use will be described where appropriate. It need not be preceded by a space but must be followed by one unless the following element is one which need not be preceded by a space, e.g. another right parenthesis.

Basic elements invented by the COBOL programmer

1 LITERALS

(a) Non-numeric literals

(i) A non-numeric literal is a string of characters bounded at each end by a quotation mark, that is the symbol ["]

(ii) The characters may be any of those in the 1900 Series COBOL alphanumeric data character set (see Appendix 1) except the quotation mark.

(iii) There must not be more than 120 characters in a non-numeric literal.

(iv) The value of a non-numeric literal is the whole string of characters between the quotation marks.

(v) The compiler takes the first quotation mark which occurs in the program to be the left bound of the first non-numeric literal, the second quotation mark to be its right bound, the third quotation mark to be the left bound of the second non-numeric literal, and so on.

(vi) Any string of characters between quotation marks is a non-numeric literal even though without the quotation marks it might be a different basic element of the language.

(b) Numeric literals

(i) A numeric literal is composed of characters from the set consisting of the numerals 0 to 9, the plus sign [+] , the minus sign [-] , and the decimal point [.] .

(ii) It must contain at least one numeric digit and if [+] or [-] appears, it must appear only once as the first character.

(iii) No more than one decimal point can appear and it must not be the last character.

(iv) The value of a numeric literal is the algebraic quantity represented by its character; it must not have more than 120 characters. It is further restricted if it is to be used in arithmetic (see Chapter 6).

2 USER-DEFINED NAMES

(a) Data names

(i) A data name is a name invented by the COBOL programmer representing data which may vary when the translated COBOL program is run.

(ii) A data name must not be a reserved word (see page 16).

(iii) A data name is composed from the set

0 to 9

A to Z

hyphen -

(iv) A data name must not exceed 30 characters in length.

(v) At least one character must be alphabetic and the name must neither begin nor end with a hyphen.

(b) Condition names

(i) A condition name is a name invented by the COBOL programmer to represent a condition. The ways of defining and using it are described on page 63.

(ii) A condition name is invented using the same rules as for data names.

(c) Procedure names

(i) The section names and paragraph names in the Procedure Division are invented by the programmer and are known as procedure names.

(ii) A procedure name is either a name invented using the same rules as for data names or a string of numeric characters (i.e. from the set 0 to 9) with a maximum length of 30 characters.

(iii) Two procedure names are the same only if they are composed of the same characters in the same order. For example, 000123 is not the same as 123.

(iv) Every paragraph name and section name must be unique within a program and must not be the same as any data name or condition name.

Basic elements treated as reserved words by the compiler

Some strings of characters or symbols always have the same meaning to the compiler. They are of two kinds: Key words and symbols, optional words.

KEY WORDS AND SYMBOLS

These are words and symbol strings which, to the compiler, have built-in meanings and which are used by the programmer to make definitions and indicate operations to be performed in the program.

Examples are ADD [>]

A full list of key words is given in Appendix 2.

OPTIONAL WORDS

These are words which may be written in the program text to make it more like English. They have no meaning to the compiler which ignores them apart from printing them in the listing of the COBOL program. They are included in the list in Appendix 2.

Optional words are given in the formats, which show what the contents of sentences may be, later in this manual. They are shown in capitals not underlined. This conforms with the normal practice in COBOL reference manuals and for compatibility with other COBOL compilers, including future I.C.T. compilers, they must be used only where shown. However, with a few exceptions, the compiler takes no account of the context in which an optional word occurs. It simply ignores it. The few exceptions are words which are key words in some places and may be used as optional words in others. Such words are specially marked in Appendix 2 and must only be used where they are shown in the formats.

KEY WORDS WHOSE USE IS NOT CONFINED TO ONE DIVISION

The meanings of most key words are determined by the division in which they occur. They are therefore described in the chapter dealing with the relevant division. Those which have the same meaning irrespective of division and can be used in more than one division are described below.

FIGURATIVE CONSTANTS

{ ZERO ZEROS ZEROES }	represents the value zero or one or more of the character 0 depending on the context.
{ SPACE SPACES }	represents one or more spaces depending on the context.
{ UPPER-BOUND UPPER-BOUNDS HIGH-VALUE HIGH-VALUES }	represents one or more of the character [←] (which consists of six binary '1's) depending on the context.
{ LOWER-BOUND LOWER-BOUNDS LOW-VALUE LOW-VALUES }	represents one or more of the character 0 depending on the context.
{ QUOTE QUOTES }	represents one or more of the character [''] depending on the context.

ALL { non-numeric literal / figurative constant } represents one or more of the string of characters represented by the non-numeric literal or the figurative constant. When the context requires a string whose length is not an exact multiple of the literal, the string of characters generated is truncated at the right-hand end. For example, if ALL "NO-OP" is used in context with a field 12 characters long, it is equivalent to "NO-OPNO-OPNO". The context must be such as to indicate the length.

When any figurative constant is associated with another data item, as when it is moved to or compared with another data item, the string of characters specified by the figurative constant is repeated on the right until its size is equal to the size in characters of the associated data item.

When a figurative constant is not associated with another data item, as when it is displayed, it represents one character. ALL cannot be used in this way.

KEY WORDS USED FOR QUALIFYING

When the COBOL programmer defines a name it need not always be unique within the program provided that when it is referred to elsewhere it can be qualified in a way which makes clear the definition referred to.

A data name can be qualified by a data name which is the name of a higher member of the same data hierarchy (see Chapter 5). The qualifier can be a qualified name. The format for this is

data-name { OF / IN } { data-name / qualified data-name }

A condition name defined in the Data Division is associated with a particular data name. The condition name can be qualified by that data name or by any name which can qualify that data name (see page 63). The format for this is

condition name { OF / IN } { data-name / qualified data-name }

An unnecessary qualifier which is legitimate according to the rules given above is permitted.

If a name is defined more than once in the program, it must be qualified whenever it is referred to. The computer will indicate duplicate names but assume the last defined.

COBOL PROGRAM SHEET

A COBOL program is written on a COBOL program sheet which is divided into 80 columns. The columns are used in the following way:

Columns 1 to 6	Sequence number
Column 7	Continuation indicator
Columns 8 to 72	Text of the program
Columns 73 to 80	Identification

Sequence number

1 If the COBOL programmer writes sequence numbers, the compiler will check that the sequence number of each line is greater than that of the previous one. Spaces within sequence numbers are converted to zeros before the check is made; for example, 000234 and ▽▽▽234 are equivalent. A message is printed by the compiler about a line whose sequence number fails the check. The line is, however, accepted as part of the program.

2 If the programmer does not assign sequence numbers, they are assigned by the compiler and will appear in the COBOL program printout produced by the compiler.

3 The compiler treats the program as 'sequenced' if the programmer gives the first line a sequence number and as unsequenced if he does not do so.

Continuation indicator

1 A hyphen in this column means that a word of the program is written partly on the previous line and partly on the line containing the hyphen. If the word is a non-numeric literal, that part of the word on the previous line continues up to and including column 72 and the continuation is preceded by a quotation mark which may be in column 12 or a later column. The first character after the quotation mark is the start of the continuation of the non-numeric literal. (For literals see page 15.)

2 If the word is not a non-numeric literal, the part of the word on the previous line must not include any spaces at the right-hand end of the line and the continuation must start in column 12.

3 If there is no hyphen in column 7, a new word starts on the new line and a character in column 72 of the previous line is considered to be followed by a space.

Program text

1 The text of the program contains headings and sentences. A heading starts in column 8. A sentence starts in column 12 or later in all divisions of the program except the Data Division in which it may start in column 8 or in column 12 or later. If a sentence continues on a new line, the continuation starts in column 12 or later.

2 A heading announces the part of the COBOL program which is to follow as a division, or within a division it announces the part which is to follow as a section or a paragraph.

Identification columns

If columns 73 to 80 of the first line of the COBOL program contain other than blanks, columns 73 to 80 of each subsequent line are compared with them. If they do not match, a message is printed by the compiler but the line is accepted as part of the program. If columns 73 to 80 of the first line contain blanks, no checks are made against columns 73 to 80 of subsequent lines and the programmer may use the columns for comments. In either case, the identification is printed as part of the COBOL program printout but does not affect the object program produced.

NOTATION

Throughout this manual the following notation is used to describe the format of COBOL statements:

1 All words printed in capital letters which are underlined must always be present when the functions of which they are a part are used. An error printout will occur during compilation if the underlined words are absent or incorrectly spelled. The underlining is not necessary when writing a COBOL source program.

2 All words printed in capital letters which are not underlined are used for readability only. They may be written or not as the programmer wishes.

3 All words printed in small letters are generic terms representing names which will be devised by the programmer.

4 When material is enclosed in braces { }, a choice must be made from the options within them.

5 When material is enclosed in square brackets [ ], it is an indication that the material is an option which may be included or omitted as required.

6 Three full stops, thus ..., indicates that any number of items may be included in a format.

Chapter 3 Identification division

The Identification Division identifies the source program and the output of a compilation. In addition, the user may include other information under the paragraphs in the following format:

FORMAT

IDENTIFICATION DIVISION.
PROGRAM-ID. Program-name.
[AUTHOR. Comment-paragraph.]
[INSTALLATION. Comment-paragraph.]
[DATE-WRITTEN. Comment-paragraph.]
[DATE-COMPILED. Comment-paragraph.]
[SECURITY. Comment-paragraph.]
[REMARKS. Comment-paragraph.]

NOTES

1 The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a full stop and a space.

2 For COBOL programs which are not subroutines, the program-name must be six characters in length. The first character must be alphabetic but the next three characters may be alphabetic or numeric. These first four characters identify the object program at run time. On a multiprogramming machine all input and output messages on the console typewriter are prefixed by these four characters. The last two characters must be numeric and are used as the priority number when the object program is being run on a multiprogramming machine.

3 For COBOL subroutines, the program-name may be between one and eleven characters in length. These characters identify the subroutine at consolidation time. The name is used to reference the Subroutine from other segments.

4 Any other information following the sentence in which the program-name occurs and preceding the next division heading is printed as part of the Source program printout but has no effect on the object program.

5 The date on which the program is compiled is always printed by the COBOL compiler and no special provision need therefore be made by the programmer to request it.

Chapter 4 Environment division

The Environment Division specifies the configuration of the compiling computer and object computer. In addition, information relating to input-output control, special hardware characteristics and control techniques can be given.

The division consists of two sections, the Configuration Section and the Input-Output Section and the Input-Output Section. The Configuration Section deals with the characteristics of the source and object computers. It is divided into three paragraphs, the Source-Computer Paragraph, the Object-Computer Paragraph and the Special-Names Paragraph. The Input-Output Section deals with the information needed to control the transmission and handling of data between external media and the object program. It is divided into two paragraphs, the File-Control Paragraph and the I-O-Control Paragraph.

The general outline of the sections and paragraphs in the Environment Division is shown in the following format which defines the order of presentation in the source program.

FORMAT

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        source-computer entry.
OBJECT-COMPUTER.        object-computer entry.
[SPECIAL-NAMES.         special-names entry.]
INPUT-OUTPUT SECTION.
FILE-CONTROL            file-control entry.
[I-O-CONTROL.           input-output-control entry.]
```

CONFIGURATION SECTION

The SOURCE-COMPUTER paragraph

FUNCTION The SOURCE-COMPUTER paragraph describes the computer upon which the program is to be compiled.

FORMAT 1

```
SOURCE-COMPUTER. COPY library-name [REPLACING word-1 BY
                 word-2 [word-3 BY word-4 ] . . . ] .
```

FORMAT 2

```
SOURCE-COMPUTER. ICL-190n.
```

NOTES

1 Format 1 is used when the COBOL library contains the entire description of the SOURCE-COMPUTER configuration. (For COPY see Chapter 8 on page 107.)

2 If format 2 is used, n in the word ICL-190n specifies the particular processor in the 1900 series on which the program is to be compiled.

The OBJECT-COMPUTER paragraph

FUNCTION The OBJECT-COMPUTER paragraph describes the computer on which the program is to be executed.

FORMAT 1

```
OBJECT-COMPUTER. COPY library-name [REPLACING word-1 BY
                 word-2 [word-3 BY word-4 ] . . . ] .
```

FORMAT 2

```
OBJECT-COMPUTER. ICL-190n
    [MEMORY SIZE integer WORDS]
    [SEGMENT-LIMIT IS integer]
    [ASSIGN OBJECT-PROGRAM TO { CARD-READER  } ] .
                              { PAPER-READER }
                              { TAPES        }
                              { EDS          }
                              { CORE-EDS     }
```

NOTES

1 Format 1 is used when the COBOL library contains the entire description of the OBJECT-COMPUTER configuration. (For COPY see Chapter 8 on page 107.)

2 If Format 2 is used, n in the word ICL-190n specifies the particular processor in the 1900 series on which the object program is to be run.

3 The MEMORY size is the amount of storage available to the object program excluding the storage required for Executive. If the object program requires more storage than the figure quoted in this clause, message 645 COMPUTER SIZE EXCEEDED (see pages 203 and 270) is printed, but the object program can still be run provided sufficient store is available for it.
The compiler will not compile a program which requires more than 32,768 words of storage. (However, see Chapter 7 concerning segmentation.)
In some instances the code generated, e.g. for MOVE statements, will depend on the object computer specified.

4 The SEGMENT-LIMIT clause specifies that sections with priority numbers less than nn are held in the permanent area of storage. The number must be a numeric literal representing an integer in the range 1 to 49. (See Chapter 7 for a full explanation of segmentation.)

5 The ASSIGN OBJECT-PROGRAM clause is used to specify to the compiler the medium from which segmented object programs are to be overlayed. If the object program is not overlayed, the clause is printed but is otherwise ignored. If TAPES is specified, the Magnetic Tape Overlay package is incorporated; if EDS, the Exchangeable Disc Overlay package. If CARD-READER or PAPER-READER is specified or if the clause is omitted, the overlay package appropriate to the medium on which the compiler resides is incorporated in the object program. The directives to the compiler (see page 185) specify the peripheral device on which the compiler is to produce the object program.

If CORE-EDS is specified an overlay package will be incorporated which assumes that the object program will be held on E.D.S. at load-time. When the program is loaded, all the overlay units that have a priority number (see page 105) less than 25 will be read into the upper part of the store, which may extend beyond 32K words if necessary. During the run, overlays are brought down from the upper part of the store into the program's overlay area as required. Overlay units with priority greater than 25 remain on E.D.S. and are overlaid from disc in the usual manner. If insufficient core is available to hold all the units with priority in the range 0 to 25, then the program will halt during loading with the message

HALTED:- NEEDS CORE

The operator should make more core available if possible; otherwise the program must be recompiled giving fewer units priority in the range 0 to 25.

B

The SPECIAL-NAMES paragraph

FUNCTION The SPECIAL-NAMES paragraph assigns programmer devised mnemonic names to the ICL hardware units.

FORMAT 1

```
[SPECIAL-NAMES. COPY library-name [REPLACING word-1 BY
                word-2 [word-3 BY word-4 ] . . . ] .
```

FORMAT 2

```
[SPECIAL-NAMES. [hardware-name integer [TYPE type-number] IS mnemonic-
name] . . .

                [CHANNEL-n IS mnemonic-name] . . .
                [*DATE IS mnemonic-name] [*TIME IS mnemonic-name]
                [hardware-name { ON STATUS IS condition-name
                               { OFF STATUS IS condition-name
                [ OFF STATUS IS condition-name ] }
                [ ON STATUS IS condition-name ] } ]...
                [DECIMAL-POINT IS COMMA] .]
```

NOTES

1 This paragraph can be omitted if its contents are not needed. If needed, it consists of one or more sentences following the heading and consisting of one or more clauses with the above formats.

2 Format 1 is used when the COBOL library contains the entire description of all the SPECIAL-NAMES used in the program. (For COPY see Chapter 8 on page 107.)

3 In format 2 the hardware name must be one of the following names:

CARD-READER

CARD-PUNCH

PRINTER

PAPER-READER

PAPER-PUNCH

4 The integer written after the hardware name enables the programmer to ensure that in certain circumstances the device is or is not the device assigned to a file. (See ACCEPT page 72 and DISPLAY page 80. The way in which the physical device to be used is selected is described on page 247.)

5 The TYPE option enables the programmer to specify a particular model of the peripheral device. The type number must be written after the word TYPE. A list of type numbers acceptable to the compiler is given on page 28. If the TYPE option is omitted and the computer on which the program is run has more than one type of the peripheral device named available, the type is assigned arbitrarily.

6 A mnemonic name is a name defined by the COBOL programmer using the same rules as for data names. It must be unique within the program and is a pseudonym for the hardward device identified in the same clause. It is used when referring to the relevant device in the Procedure Division. (See ACCEPT on page 74 and DISPLAY on page 80.)

7 With *DATE and *TIME it is possible to obtain from Executive the date and time respectively by use of the ACCEPT statement (see page 74).

8 CHANNEL-n represents one of the channels on the tape loop used to control the line spacing of printers. n must be an integer in the range 1 to 7 and refers to the corresponding tape channel. The mnemonic name is a psuedonym for the particular channel and is used in the WRITE statement (see page 103). If CHANNEL-n is assigned a mnemonic name, that name applies to channel n of every tape loop involved in the program.

9 In the STATUS clause the hardware device named must be a switch. The condition name indicates the ON or OFF state of the switch, according to the form of the clause used and can be used in the Procedure Division wherever a condition is permitted. If the ON state is named, the value of the condition is "true" whenever the switch is on and "false" whenever it is off. If the OFF state is named, the reverse is true.

10 The switches whose ON and OFF states can be named are called BIT-nn where $00 \leq nn \leq 23$. nn corresponds to bits 0 to 23 of word 30 of the object program. A 1-bit value corresponds to ON, and a zero bit value to OFF. The ON or OFF or both states of the switch may be named in one clause. The switches can be set by operator action, details of which are given on page 192.

11 The clause DECIMAL-POINT IS COMMA means that the functions of comma and period are exchanged in the PICTURE clause character-string and in numeric literals. (See pages 139 and 140.)

INPUT-OUTPUT SECTION

The FILE-CONTROL paragraph

FUNCTION

The FILE-CONTROL paragraph names each file, identifies the file medium and allows particular hardware assignments. The paragraph also specifies alternative input-output areas.

FORMAT 1

```
FILE-CONTROL. COPY library-name [REPLACING word-1 BY
                word-2 [word-3 BY word-4 ] . . . ] .
```

FORMAT 2

```
FILE-CONTROL.
[SELECT file-name-1 ASSIGN TO { [integer-1] TAPES                          }
                              { TAPES [integer-2]                          }
                              { hardware-name integer-3 [TYPE type-number] }
[FOR MULTIPLE REEL]
[RESERVE { 1  } ALTERNATE [ AREA  ]]
         { NO }           [ AREAS ]
[ACCESS MODE IS [ SEQUENTIAL ]]
               [ RANDOM     ]
[PROCESSING MODE IS SEQUENTIAL]
[{ORGANIZATION} IS { INDEXED      [FIRST-LEVEL] [SECOND-LEVEL] }]
 {ORGANISATION}    { DIRECT       [FIRST-LEVEL] [SECOND-LEVEL] }
                   { FIRST-LEVEL  [SECOND-LEVEL]               }
                   { SECOND-LEVEL                              }
[ACTUAL KEY IS data-name-1]
[SYMBOLIC KEY IS data-name-2 ] .]
[SELECT file-name-2...]...
```

NOTES

1 Format 1 is used when the COBOL library contains the entire description of the FILE-CONTROL paragraph. (For COPY see Chapter 8 on page 107.)

2 A file name in format 2 is a data name and is therefore invented according to the rules for data names (see page 15).

3 Every file of information referred to in the Procedure Division must appear in a SELECT .... ASSIGN statement, since this is the way of indicating the peripheral device used to hold the file. This includes sort files, which must, however, be ASSIGNed to magnetic tape, E.D.S., or F.D.S.

4 integer-1 indicates the number of magnetic tape units to be assigned to the file. In accordance with the 1900 Series operating systems, however, it is not necessary to allocate more than one magnetic tape unit to a file. When the end of a reel is sensed during the object program run, the continuation reel is sought on all other units. If it is not found, the object program is suspended and instructions are given to the operator to mount the new reel and restart the program. The allocation of more than one tape to a file therefore has no effect. The facility is provided only for compatibility and documentation purposes.

"integer-2" (which must be in the range 1-15), is the unit number to be allocated to this file. Details are essentially the same as those in Note 6 below, as applied to Mass Storage devices. This facility for controlling the allocation of device numbers may be necessary when compiling subroutines to ensure that different subroutines in a program do not access the same unit. As with Mass Storage devices, it is permissible for files to have the same unit number so long as they are not open at the same time. The programmer should note that for any files not assigned to "TAPES integer-2" in the File-Control paragraph unit numbers will still be allocated consecutively starting at 1. For example, if the first two magnetic tape SELECT...ASSIGN statements in the File-Control paragraph are "...ASSIGN TO TAPES 1" followed by "...ASSIGN TO TAPES" (or vice versa), both files will be allocated unit number 1.

5 The hardware devices other than magnetic tape units to which files can be assigned have the following names:

CARD-READER

PAPER-READER

PRINTER

CARD-PUNCH

PAPER-PUNCH

EDS (Exchangeable Disc Store)

FDS (Fixed Disc Store)

MCF (Magnetic Card File)

No differentiation is made in the COBOL language between E.D.S. and T.E.D.S. COBOL users with T.E.D.S. should refer to it as EDS in all relevant places. For example.

SELECT file-name ASSIGN EDS n .

6 The integer written after the hardware name identifies the particular device to the programmer and gives him some control over its allocation. When a file is opened in the object program and is to be allocated to a device in the list given above, the allocation is done dynamically in that the file is assigned to the next available device, although the operator can override the automatic choice. The device chosen in this way becomes the program's integer-2 device of that type, where integer-2 is the number assigned to it in the SELECT....ASSIGN statement and is less than or equal to 63. For example

SELECT file-name ASSIGN PRINTER 4

does not mean that the file is to be assigned to the fourth printer but merely that it is to be assigned to a printer which the program refers to as number four. With slow peripherals the same peripheral device can be used for more than one purpose at the same time if two files are assigned to the same type of device with the same number or, if an ACCEPT or DISPLAY statement refers to the same device while the file is open. However, since the device is released from the program when either of the files is closed or an ACCEPT is completed (except in the case of a direct acces file). it is unwise to do so.

7 The TYPE option in a SELECT....ASSIGN statement enables the programmer to specify that a particular model of a character peripheral device is to be assigned to a file. The number written after the word TYPE is equivalent to the property code described in the 1900 Series System Manual. It indicates that the unit to be assigned must have at least the properties defined by the property code, even though the unit actually assigned may be capable of more than the required action, for example a greater speed.

The property code is understood by the Executive program at run time as a string of nine bits. The first bit is set to zero if the unit assigned must have at least the properties specified in the following eight bits and to one if it must not have the following properties. The other eight bits specify the properties and have meanings defined according to the kind of advice.

The number written by the COBOL programmer is the octal equivalent of the nine bit string, i.e. a three digit number. For example, the octal equivalent of 010001100 is 214, formed by writing a number to represent each of the three bit strings in turn. (An octal number in PLAN is preceded by the character # but this is not so in this case. The fact that the number is octal can usually be ignored. It is explained to allow the programmer to relate the above information to that given in the 1900 Series System Manual.)

If more than one recording mode is possible on a particular input/output medium such as paper tape, a device which can handle the particular recording mode must usually be assigned by means of the TYPE option. The recording mode must be defined either by the RECORDING MODE clause (see page 36) or by operator action.

The TYPE option can be used only for character peripheral devices. The type numbers and the devices they represent are as follows:

Type number	Unit number	Description
CARD READERS		
110	1911	900 cards a minute with binary image facility
100	1911	900 cards a minute without binary image facility
010	1912	300 cards a minute with binary image facility
000	1912	300 cards a minute without binary image facility
020	2102	300 cards a minute
CARD PUNCHES		
040	1920	100 cards a minute
000	1922	33 cards a minute
140	2151	300 cards a minute
PAPER-TAPE READERS		
000	1915	300 characters a second
040	1916	1000 characters a second
000	2601	300 characters a second
040	2602	1000 characters a second
PAPER-TAPE PUNCHES		
000	1925	110 characters a second
000	2601	110 characters a second
000	2602	110 characters a second
LINE PRINTERS		
150	1930	1000 lines a minute and 120 print positions
001	1931	300 lines a minute and 96 print positions
011	1931	300 lines a minute and 120 print positions
041	1932	600 lines a minute and 96 print positions
051	1932	600 lines a minute and 120 print positions
145	1933	1350 lines a minute and 96 print positions
155	1933	1350 lines a minute and 120 print positions
175	1933	1350 lines a minute and 160 print positions

8 The clause for MULTIPLE REEL appears in the COBOL program printout if it is written but it has no effect on the object program since the possibility of a file occupying more than one reel of magnetic tape is always allowed for.

9 The clause RESERVE....ALTERNATE AREAS can be used to control the number of buffer areas used in transferring information between the peripheral device assigned to a file and the computer core store. If the clause is omitted or NO ALTERNATE AREAS is specified, a standard number of areas is assigned. If 1 ALTERNATE AREA is requested, more storage is allocated to the file and the rate of processing it may be considerably improved. The allocation of more than one

alternate area is not permitted since it makes no improvement. RESERVE 1 specified for a file held on a direct access device has the effect of allocating one buffer for the home bucket and another for the overflow bucket.

10 The ACCESS MODE clause indicates the manner in which the records of a direct access file are read or written. If the clause is omitted, ACCESS MODE IS SEQUENTIAL is assumed.

If ACCESS MODE IS SEQUENTIAL is specified for a file stored serially, the records are accessed in the order in which they are stored. The effects of DELETE, READ, REWRITE and WRITE statements on such a file are as follows:

(a) A DELETE statement deletes the last record read. The file must have been opened for input-output and the DELETE statement must be preceded by a READ statement.

(b) A READ statement makes available the logical record in the next physical position in the file.

(c) A REWRITE statement replaces the last record read by the record specified. The file must have been opened for input-output and the REWRITE statement must be preceded by a READ statement.

(d) On the execution of a WRITE statement the following two cases can occur:

(i) If the file has been opened for output, a WRITE statement places the specified record in the next physical position of the file.

(ii) If a file has been opened for input-output, a WRITE statement causes the specified record to be inserted in the physical position immediately preceding the last record read.

If ACCESS MODE IS SEQUENTIAL is specified for a file whose records are in key-sequence order, the records are accessed in key-sequence order. The effects of DELETE, READ, REWRITE and WRITE statements on such a file are as follows:

(a) A DELETE statement deletes the last record read. The file must have been opened for input-output and the DELETE statement must be preceded by a READ statement.

(b) A READ statement makes available the next record in key-sequence order.

(c) A REWRITE statement replaces the last record read by the record specified. The file must have been opened for input-output and the REWRITE statement must be preceded by a READ statement.

(d) A WRITE statement causes the specified record to be inserted in the logical position immediately preceding the last record read. The file must have been opened for input-output and the WRITE statement must have been preceded by a READ statement.

The ACTUAL KEY clause must not be specified for the SEQUENTIAL mode.

11 ACCESS MODE IS RANDOM must be specified when it is required to reference *specific* records of a file. If it is specified, the SYMBOLIC KEY and ORGANIZATION clauses must also be specified.

If ORGANIZATION IS DIRECT is specified (that is, the file is stored randomly), each record is obtained or placed randomly, that is, a specific logical record is made available from the file on the execution of a READ statement or is placed in a specific location on the file on the execution of a WRITE or REWRITE statement. If ORGANIZATION IS DIRECT is specified the ACTUAL KEY clause must also be specified for the file.

ORGANIZATION IS INDEXED indicates that the records of the file are stored in key-sequence order and standard indexes are created and maintained with the file. If ORGANIZATION IS INDEXED is written records can be retrieved in two ways: either a specified record is retrieved or a group of records in key-sequence order starting at a specified record is retrieved. Specific records are retrieved by obeying a SEEK statement followed by a READ statement. The record retrieved is the record identified by the contents of the SYMBOLIC KEY field. A group of records beginning at a specific record is accessed by retrieving the first of the group in the manner described above; subsequent records are accessed by obeying READ statements not preceded by SEEK statements. The ACTUAL KEY clause must not be written for a file for which ORGANIZATION IS INDEXED is specified.

12 The PROCESSING MODE IS SEQUENTIAL clause specifies that records of direct access files are processed in the order in which they are accessed. The clause is optional and is included for documentation purposes.

13 The ORGANIZATION clause specifies the data organization of files which are held on direct access devices and can be accessed in the random mode.

ORGANIZATION IS INDEXED indicates that the position of each logical record in a file is determined by indexes created and stored with the file. Records of a file whose organization is INDEXED are stored in key-sequence order with standard index-tables created and maintained by the 1900 system. The SYMBOLIC KEY clause must be specified if ORGANIZATION IS INDEXED is specified. The ACTUAL KEY clause, however, must not be specified.

If specific records of a file whose organization is INDEXED are being processed each DELETE, READ or WRITE statement must be preceded by a SEEK statement. The effect of the SEEK statement is to transform the contents of the SYMBOLIC KEY field into a logical bucket number which in this case is held within the direct access system and is not available to the programmer.

When preceded by a SEEK statement without an intervening READ statement the effects of DELETE, READ and WRITE statements on a file whose organization is INDEXED are as follows:

(a) A DELETE statement deletes the record which is specified by the SYMBOLIC KEY field and held in the bucket determined by the preceding SEEK statement.

(b) On a READ statement the direct access system searches the bucket whose logical bucket number has been determined by the preceding SEEK statement for the record specified by the SYMBOLIC KEY field.

(c) A WRITE statement inserts the record specified in the statement (in key-sequence order) in the bucket determined by the preceding SEEK statement in the position in the bucket determined by the SYMBOLIC KEY field.

When processing such a file, the effects of DELETE, READ, REWRITE and WRITE statements preceded by a READ statement without an intervening SEEK statement are the same as for a file whose access mode is SEQUENTIAL and whose records are stored in key-sequence order (see page 29).

ORGANIZATION IS DIRECT indicates that the records of a file are stored in buckets whose logical bucket numbers are determined from the keys by an address generation procedure. The effects of DELETE, READ, REWRITE and WRITE statements on a file whose organization is DIRECT are as follows:

(a) A DELETE statement deletes the record which is specified by the SYMBOLIC KEY field and held in the bucket specified by the ACTUAL KEY field.

(b) On a READ statement the direct access system searches the bucket which is specified in the ACTUAL KEY field for the record with the key specified in the SYMBOLIC KEY field.

(c) A REWRITE statement replaces the last record read from the file by the record specified in the REWRITE statement.

(d) A WRITE statement inserts the record specified in the statement at the end of the bucket specified by the ACTUAL KEY field.

FIRST-LEVEL indicates that the Direct Access Housekeeping Package is required to handle any first-level overflow that occurs on the execution of a WRITE or REWRITE statement to the file.

SECOND-LEVEL indicates that the Direct Access Housekeeping Package is required to handle any second-level overflow that occurs on the execution of a WRITE or REWRITE statement to the file.

It should be noted that the Package can always access records that are already in overflow areas; thus it is not necessary to use the clauses FIRST-LEVEL or SECOND-LEVEL for a file to which only READ or DELETE statements are to be applied.

Both the SYMBOLIC KEY and the ACTUAL KEY clauses must be specified for a file whose organization is DIRECT.

14 The ACTUAL KEY clause specifies the logical bucket number of the bucket in which a record is to be placed or from which it is to be retrieved. The clause must be present for a file whose access mode is RANDOM and whose organization is DIRECT. The contents of data-name-1 are used by the DELETE, READ and WRITE statements to locate a specific direct access record. Prior to the execution of one of these statements, therefore, the logical bucket number of the record whose key is specified by the SYMBOLIC KEY field must be placed in data-name-1 by the application of the appropriate address generation procedure. The contents of data-name-1 remain unchanged unless explicitly changed in the source program.

Data-name-1 must be a numeric integral field less than 7 characters in length if described as DISPLAY, DISPLAY-3 or COMPUTATIONAL and less than 24 binary digits in length if described as COMPUTATIONAL-1.

The ACTUAL KEY clause must not be specified for files whose access mode is SEQUENTIAL or whose organization is INDEXED.

15 The SYMBOLIC KEY clause must and can only be specified for a file whose access mode is RANDOM. For such a file data-name-2 specifies the field in which the key of the desired record is to be placed before a DELETE, READ, REWRITE, SEEK or WRITE statement is executed for the file.

Data-name-2 must be a character field and may be alphabetic, numeric or alphanumeric. It must have the same length as the fixed key length defined for the file when it is allocated and it may not be more than 64 characters in length. It remains unchanged unless explicitly changed in the source program.

If ORGANIZATION IS INDEXED is specified, data-name-2 is transformed into the logical bucket number of the specified record by means of a SEEK statement, which is followed by the input/output statement it is desired to apply to the record.

If ORGANIZATION IS DIRECT is specified, before executing any input/output statement the programmer must explicitly transform the key contained in data-name-2 into the logical bucket number which is placed in the ACTUAL KEY field (data-name-1) by applying the appropriate address generation procedure.

The I-O-CONTROL paragraph

FUNCTION The I-O-CONTROL paragraph specifies the input-output techniques, the points at which rerun is to be established and the storage area which is to be shared by different files.

FORMAT 1

```
[I-O-CONTROL. COPY library name [REPLACING word-1 BY word-2
[word-3 BY word-4 ] ... ] .
```

FORMAT 2

```
[I-O-CONTROL.
[APPLY data-name-1 TO TRANSFER-REPLY ON file-name-1]
[APPLY data-name-2 [, data-name-3] TO INDEX-BUFFER ON file-name-2]
[APPLY SENTINEL-PROCESSING ON file-name-3]
[APPLY FILE-MESSAGES ON file-name-4 [file-name-5 . . .] ]
[APPLY data-name-4 TO SORT-* ON file-name-6 [,file-name-7] . . . .]
[RERUN [ON { file-name-8 }  ]]   EVERY { integer-1 RECORDS OF file-name-9 }  ]]
           { TAPE-*      }               { END OF REEL OF file-name-9     }
           { EDS-*       }
           { FDS-*       }
           { MCF-*       }
[SAME [RECORD] AREA FOR file-name-10 [file-name-11 . . . ]] ]
```

NOTES

1 The I-O CONTROL paragraph can be omitted if its contents are not needed.

2 Format 1 is used when the COBOL library contains the entire I-O-CONTROL paragraph. (For COPY see Chapter 8 on page 107.)

3 The APPLY . . . TO TRANSFER-REPLY clause causes the input-output system to place in data-name-1 reply information arising from the transfer or attempted transfer of a record to or from file-name-1.

Data-name-1 must not be defined in the file to which it is to be applied. It must be a binary field described as PICTURE 1(24) SYNCHRONIZED RIGHT. The meaning of the values of data-name-1 is given below.

Contents of reply word	Meaning	Context
0	No exception condition has arisen	
1	Record Missing	Occurs when an attempt is made to READ or DELETE a specific record on a file whose organization is INDEXED or DIRECT and the record is not present.
2	Failure of integrity code check	Occurs when an attempt is made to OPEN a file in a mode incompatible with the current use of the file by other programs (see Direct Access Techniques manual page 2-2-4).
3	Purge date not exceeded	Occurs if an attempt is made to OPEN a file for output or input-output when the file may not be written to.
4	Overflow error	Occurs when there is no room in the file to hold the expanded or inserted record specified in a WRITE or REWRITE statement.
5	File not in system	Occurs when an attempt has been made to open a file which is not on-line.

Contents of reply word	Meaning	Context
6	Duplicate Key	Occurs when an attempt is made to WRITE a record to a file for which ACCESS MODE IS RANDOM is specified, and a record with this key is already on the file.

If file-name-1 refers to a Printer file the reply information placed in data-name-1 will be :

Contents of reply word	Meaning
0	No exception condition
4	Transfer failure
8	Paper Low
16	Page Overflow

It is possible for some exception condition to occur together. In this case the values are summed : for example, paper low and page overflow = 8 + 16 = 24.

4 The APPLY . . .TO INDEX-BUFFER clause can only be written for files whose access mode is RANDOM and organization is INDEXED. Its effect is to reduce the number of transfers of index tables to store on executing SEEK statements. Data-name-2 and data-name-3 represents areas which are to be assigned to hold the first and second level indexes respectively. They must refer to areas which begin on a word boundary and are not described in the file to which they are to be applied. If the index table occupies less than a bucket, the corresponding area must be three words longer than the index table. If the index table occupies more than one bucket, the corresponding area must be an area equivalent in length to an integral number of buckets plus three words. The programmer must not change the contents of data-name-2 or data-name-3 between SEEK statements; if he does, the results will be unpredictable.

5 The APPLY SENTINEL-PROCESSING clause may only be used for magnetic tape files. The effect is to present in the record area the 16 words of user information in a User Sentinel, when such a sentinel is read. The record area must be at least 16 words long. In order that the programmer can detect the reading of a sentinel, monitor switch 12 in Word 30 is set.

6 The APPLY FILE-MESSAGES clause causes run-time messages (for example, LOAD FILE *nn* ON *pp*) to be produced in object programs for the slow peripheral files named file-name-4, file-name-5. The naming of a fast peripheral file will cause error message 917 to be printed at compile-time.

7 The APPLY . . TO SORT-* clause can only be written for sort-files, and is used to give information about the sort environment. Data-name-4 is the name of an area which will be 9 or 14 words long. This area must begin on a word boundary and the information in it applies to all the files following the word ON.

14-word area for direct access sort

(a)	*WORDS 1 TO 5*	Library File information
	Words 1 to 3	File-name
	Word 4	Generation number (in binary)
	Word 5	Device type 1=E.D.S., 2=Twin-E.D.S., 3=F.D.S., 4=magnetic tape.
(b)	*WORDS 6 TO 9*	Workfile 1 information
	Words 6 to 8	File-name
	Word 9	Generation number (in binary)
(c)	*WORDS 10 TO 13*	Workfile 2 information
	Words 10 to 12	File-name
	Word 13	Generation number (in binary)
(d)	*WORD 14*	Reply word

9-word area for magnetic tape sort

WORDS 1 TO 6	Library file information
Words 1 to 3	File-name
Word 4	Reel Sequence number
Word 5	Generation number (in binary)
Word 6	Device type
WORD 7	Number of work tapes (range 3 to 10)
WORD 8	Input file reel sequence number
WORD 9	Reply word

If there is no APPLY clause then the following environments are assumed:

(a) *For direct access*

Library file name $PROGRAM_{\nabla}XSXA$, Generation number = 0, Device type E.D.S.

Workfile 1 name $COB_{\nabla}WORKFILE$, Generation number = 1
Workfile 2 name $COB_{\nabla}WORKFILE$, Generation number = 2
} on E.D.S.

(b) *For magnetic tape*

Library file name $PROGRAM_{\nabla}XSXM$, reel sequence number = 0, Generation number = 0, device type = MT, number of work tapes = 3.

Use of the Reply Word

If there has been an error in the sort program and restart point 29 is used then the reply word is set to 1 and control is returned to the statement following the SORT statement even if there is an applicable output procedure.

8 The RERUN clause enables the programmer to specify that information is to be dumped periodically on a peripheral device while the program is being run. The clause may occur only once in the whole program. The information will permit the program to be restarted if necessary from the point at which a dump is made. The routines and operating instructions for doing this are supplied with the compiler and are described more fully on page 246.

If the information is to be dumped on the file whose condition controls the dumping, the ON option must be omitted and the file must be open for output when the dump is made. If the information is to be dumped on magnetic tape on a tape-deck reserved for this purpose, ON TAPE-* must be written. ON EDS-*, ON FDS-* and ON MCF-* must be used if the information is to be dumped on a direct access medium.

If the information is to be dumped on a file other than the controlling file, ON file-name-8 must be written. File-name-8 is the name of the file on which the information is to be dumped. It must be held on magnetic tape and must be open for output when the dump is made.

The program dump may be made either every time a specified number of records of the specified file have been processed, in which case the option EVERY integer RECORDS must be written, or every time the end of a magnetic tape reel of the specified file is met, in which case the option END OF REEL must be written. If the file which is controlling the dumping is a direct access file the EVERY integer-2 RECORDS option must be used.

File-name-9 specifies the file controlling the dump.

9 The SAME AREA clause is used to specify that two or more files are to use the same storage area for processing. If the RECORD option is used, the files will share only the area where the current logical record is processed and a record from only one of the files can be available at any one time. If the word RECORD is omitted, the area being shared will include any buffers used to transfer information between the peripheral devices and the internal store. In this case, not more than one of the files named in a SAME AREA clause must be open at any one time (see OPEN page 93).

Chapter 5 Data division

The Data Division is that part of the COBOL program in which the programmer

1 specifies how the internal storage available to the program is divided and gives names to the areas specified,

2 gives information about the data files which are to be presented to the computer by the program and those which the program will cause the computer to produce,

3 specifies properties of the internal storage areas defined which modify the meaning of operations involving those areas.

The division is subdivided into four sections which are the File, Working-Storage, Constant and Linkage Sections. The File Section defines the contents of data files stored on an external medium. Each file is defined by a file description followed by a record description or a series of record descriptions. The Working-Storage Section describes records and non-contiguous data items which are not part of external data files but are developed and processed internally. The Constant Section describes data items whose values are assigned in the source program and do not change during the execution of the object program. The Linkage Section describes subroutine parameters.

The structure of the Data Division is there as follows:

```
DATA DIVISION.
[FILE SECTION.
 Contents.]
[WORKING-STORAGE SECTION.
 Contents.]
[CONSTANT SECTION.
 Contents.]
[LINKAGE SECTION.
 Contents.]
```

DEFINITIONS OF TERMS

FILE A file is a stream of information which is either presented to or produced by the computer under the control of the program. It is held on some external storage medium to which information is sent or from which information is received by the computer.

The information in the file is presented to or produced by the computer a record at a time. Records are held on the external medium in blocks containing one or more records. The number of records in a block depends upon the particular external medium and the amount of storage available in the internal store since there must be enough storage for at least one block from each file. The size of a block in relation to the various peripheral devices is described in the notes on BLOCK CONTAINS on page 37.

RECORD A record is the unit of a file which is available for processing at any one time or an equivalent area of storage defined in the Working-Storage or Constant Section. It is defined in the Record Description explained on page 42. A file may contain more than one kind of record, in which case each must have a Record Description.

The definition of a file and its records in the Data Division causes internal storage to be assigned for at least one block of the file and for the longest record of the file. Each record in the file is fed into and made available for processing in the same area of storage reserved for the records.

FIELD A field is an area of storage which is defined and given a data name in the Data Division. It may be either part of a record or a separately defined area.

ELEMENTARY FIELD An elementary field is one which is not divided into subordinate fields. It must not be longer than 120 characters if in character form or 48 bits including the sign if in binary form. A record can be an elementary field.

GROUP FIELD A group field is a field which is divided into elementary fields. A record can be a group field.

DATA DESCRIPTION ENTRY A data description entry is a sentence in the Data Division defining a field or a condition.

FILE SECTION

The File Section consists of the section heading

FILE SECTION.

followed in an arbitrary order by a file definition for each file processed by the program. Each file definition consists of a file description followed by a record description for each kind of record in the file. The File Section therefore has the following structure:

```
FILE SECTION.
    file description
    record description
    record description
        .
        .
        .
    file description
    record description
    record description
        .
        .
        .
    file description
    record description
    record description
        .
        .
        .
```

If a file contains more than one kind of record, the record descriptions are written one after the other in arbitrary order. The compiler allocates a record area in storage large enough to hold the largest record defined. The declaration of several kinds of records in a file is thus a way of mapping the record area in several ways. It is the programmer's responsibility to ensure that the record in the record area at any particular time conforms to the mapping of the store assumed by the program at that time.

FD SKELETON

File description

FUNCTION The File Description furnishes information concerning the physical structure, identification and record names pertaining to a given file.

FORMAT 1 FD file-name COPY library-name [REPLACING word-1 BY word-2 [word-3 BY word-4] . . .]

FORMAT 2

```
FD file-name
[RECORDING MODE IS { F | V | 2 }]
[FILE CONTAINS ABOUT integer-1 RECORDS]
[BLOCK CONTAINS [integer-2 TO] integer-3 { RECORDS | CHARACTERS }]
[RECORD CONTAINS [integer-4 TO] integer-5 CHARACTERS]
[LABEL { RECORDS ARE | RECORD IS } { STANDARD [WITH GENERATION-NO] | OMITTED }]
[VALUE OF { ID | IDENTIFICATION } IS { data-name-1 | literal-1 }]
[ACTIVE-TIME IS { data-name-2 | literal-2 } ]
[GENERATION-NO IS data-name-3]
[DATA { RECORD IS | RECORDS ARE } data-name-4 [data-name-5 . . .]]
[SEQUENCED ON data-name-6 [data-name-7 . . .]]
```

NOTES

FORMAT 1 is used when the COBOL library contains the File Description entry. Otherwise format 2 is used. (For COPY see Chapter 8 on page 107).

FD file-name clause

The level indicator FD identifies the beginning of a File Description and must precede the file-name of the file being defined.

RECORDING MODE clause

1 The RECORDING MODE clause specifies the format or organization of data on external media.

2 RECORDING MODE F or V can only be used of magnetic tape or direct access files. If used, the compiler allocates an extra word of storage at the beginning of the record area for the file, and the object program, on obeying an output statement, will insert a count word at the beginning of each record and set the appropriate value in it.

RECORDING MODE IS F means that all the records in the file have the same fixed length.

RECORDING MODE IS V means either that the records in the file may have different fixed lengths or that one or more records may contain an OCCURS . . . DEPENDING clause.

If RECORDING MODE IS F is written in the FD for a file whose records are not in fact of the same fixed length the compiler will flag an error but will continue as if RECORDING MODE IS V had been written, thus producing an object program that can correctly process the records as described.

If the RECORDING MODE clause is not written in the FD for a file assigned to fast peripheral the compiler will assume that count-words are specifically defined in each record of the file.

3 RECORDING MODE IS 2 is used for binary image card-reading, and indicates that each column of the card will occupy two character positions of the record area. The programmer must define the record area to be large enough and must assign a card reader to the file which can read a card image (see page 28).

The omission of the RECORDING MODE clause causes the standard recording mode to be assumed.

4 The recording modes for the various external media are described on pages 127 to 130.1.

FILE CONTAINS clause

1 The FILE CONTAINS clause indicates the approximate number of logical records in a file.

2 If the clause is used, it is printed as part of the source program but has no effect on the program, being merely for documentation purposes.

BLOCK CONTAINS clause

1 The BLOCK CONTAINS clause specifies the size of a physical record.

2 If a file is assigned to *magnetic tape* and blocks may contain more than one record, the BLOCK CONTAINS clause must be used to specify the size of the block in terms of characters or records. If a file is assigned to magnetic tape and the clause is omitted, the file is assumed to contain single record blocks.

3 If the RECORDS option is used in the description of a magnetic tape file, the buffer storage allocated will be sufficient to hold integer-3 of the largest record defined in the file.

4 If the CHARACTERS option is used in the description of a magnetic tape file, the buffer storage allocated will be large enough to hold integer-3 characters.

5 The size of blocks when using industry compatible tape is restricted only by the amount of internal storage left after deducting other requirements of the program. By convention, however, magnetic tape blocks are not longer than 512 words, if the program is using less than 8192 words of storage, and 1024 words if the program has more than 16,384 words of storage. (See also page 132.)

6 If a file is assigned to a *character peripheral device,* a block always contains only one record and the BLOCK CONTAINS clause is therefore ignored if used. On *paper tape* a block is as long as the longest record of the file which must not exceed 128 characters in length. On a *buffered printer* the block size must not exceed the size of the print line. On other devices the block size is dictated by the characteristics of the device, e.g. the length of the print line, the number of card columns. In these cases, if the record is less than a block in length, the rest of the block is ignored on an input file and filled with unpredictable characters on an output file.

7 If a file is held on a *direct access device,* the clause causes the allocation to the file of a buffer area of the specified size. This size must be equal to the bucket size specified when allocating the file less the size of the bucket header.

LABEL RECORD clause

1 The LABEL RECORD clause specifies whether labels are present, and if present, identifies the label.

2 Labels must not be specified for files held on *character peripheral devices.* Such files can be described as

$$\underline{\text{LABEL}} \left\{ \begin{array}{l} \underline{\text{RECORDS}} \text{ ARE} \\ \underline{\text{RECORD}} \text{ IS} \end{array} \right\} \underline{\text{OMITTED}}$$

or the clause may be left out altogether.

3 *Direct access and magnetic tape files* must have labels and must be described as follows:

$$\underline{\text{LABEL}} \left\{ \begin{array}{l} \underline{\text{RECORDS}} \text{ ARE} \\ \underline{\text{RECORD}} \text{ IS} \end{array} \right\} \underline{\text{STANDARD}} \text{ [WITH } \underline{\text{GENERATION-NO}}\text{]}$$

The clause specifies that the file being described is accompanied by standard 1900 series labels which will be read from the tape and checked if the file is opened as an input file, or written on to the tape if the file is opened as an output file. The GENERATION-NO option specifies that the field within the label known as

the generation number (see below) is to be checked in an input file and written in an output file. The value against which it is checked or which is recorded in it is specified either by the GENERATION NO IS clause (see below) or at run-time by an ACCEPT statement (see page 74). If the option is not used the generation number is not checked for an input file and is recorded as zeros for an output file.

Since a normal part of data processing is writing an updated version of an input file which will be processed by a subsequent run of the same program, more than one file in an installation will usually have the same identification. To permit this but still avoid confusion, the generation number field acts as a qualifier to the identification and can be specified as described above during the running of the program.

VALUE OF clause

1 The VALUE OF clause gives details of items included in the labels associated with a file.

2 The form and complete contents of the labels accompanying a magnetic tape file are described in detail in the 1900 Series manual *Magnetic Tape*. The three fields in the label preceding the file which the COBOL programmer can control are as follows:

Identification (called the File Name in *Magnetic Tape*)

Active-Time (called the Retention Period in *Magnetic Tape*)

Generation-Number (called the File Generation Number in *Magnetic Tape*)

The use of the Generation-Number field has been explained on page 37. The use of the other two items is as follows.

3 IDENTIFICATION When a file is opened for processing, the system searches for a file with the identification specified if it is an input file which is to be read, or writes on to the tape or direct access medium a label containing the identification specified if it is an output file to be written.The identification may be specified as being contained in the area referenced by "data-name-1" or as "literal-1". "Data-name-1" must be a character field of up to 12 characters in length. If the field is less than 12 characters in length, the contents will be taken and space characters will be added at the right hand end. If the field is non-character or longer than 12 characters, message 924 will be printed at compile time and the file name SCRATCH TAPE assumed. "Literal-1" must be non-numeric and must not exceed 12 characters in length. The identification must then remain the same every time the program is run.

The programmer must specify an identification for each file which has a standard label.

4 The ACTIVE-TIME option specifies the number of days for which the file is to be retained and is used only if the file is to be written as an output file. It may be specified as being contained in the area referenced by "data-name-2" or as "literal-2".

"Data-name-2" must be numeric and must not exceed 6 decimal digits or 23 binary bits.

"Literal-2" must be an unsigned integral numeric literal which does not exceed 999. The system will ensure that the file produced will not be over-written for the number of days corresponding to the number stated. If the value of the active-time is not specified, an active-time of zero is assumed, implying that the file is not needed after the program has been run. In this case the file would presumably be read as input by the program which produced it. An active-time of zero should never be given on a multiprogramming machine in the 1900 Series since, when such a file has been closed, its tape may be used by another program using the machine at the same time.

5 The GENERATION-NO IS clause is used in conjunction with the WITH GENERATION-NO option in the LABEL RECORDS clause (see above). Data-name-3 must be the name of a numeric field of not more than 6 decimal digits or 23 binary bits. The number contained in data-name-3 will be used, on input, to check the generation number of a tape label, and on output, to create a generation number in a tape label.

When this method is used to supply or check generation numbers it is not necessary to user the ACCEPT FILE KEYS statement for any files other than direct access files being opened in output mode (see page 143).

DATA RECORDS clause

1 The DATA RECORDS clause lists the names of the kinds of records which appear in a file.

2 The compiler prints the clause as part of the source program but does not use it in any way.

SEQUENCED ON clause

1 The SEQUENCED ON clause lists the fields in the records on which the records are sequenced.

2 The compiler prints the clause as part of the source program but does not use it in any way.

Sort file description

FUNCTION The Sort File Description gives information about the size of a block of data in the Sort-file. It is followed, just as a File Description is followed, by a description of its records.

FORMAT SD file-name

BLOCK CONTAINS integer { RECORDS / CHARACTERS }

NOTES

1 SD file-name clause: the level indicator SD identifies the beginning of a Sort File Description and must precede the file-name of the file being defined.

2 File-name must be ASSIGNed to magnetic tape, EDS or FDS. The assigned hardware device establishes the file medium for the sort (excluding the library file).

3 BLOCK CONTAINS clause: used to allocate a buffer area to the sort-file according to the rules on page 37.

For a direct access sort this area must be large enough to hold a workfile bucket. (The two workfiles must have the same bucket size.)

For a magnetic tape sort the clause determines the block size of the file created by the input procedure.

4 The name Sort File merely designates a set of records to be sorted by the SORT statement, i.e. file-name is not a physically existing file, which is why only the BLOCK CONTAINS. . . . clause is required.

The file that is actually sorted is either the input file to the sort or the workfile containing the records given by the input procedure.

5 The other clauses allowed in a File Description are acceptable in a Sort File Description, but are unnecessary. They will be checked for syntactical errors in the normal way, but ignored.

WORKING STORAGE SECTION

The Working Storage Section is used to define areas of storage which are needed during the running of a program but which do not form part of any file. Both elementary and group fields can be defined as Working Storage areas.

The section has the following structure:

WORKING-STORAGE SECTION.

data description entry.

data description entry.

.

.

.

The following points should be noted about the use of the section:

1 Each field defined must have a unique name unless it is a subfield of another field.

2 A field which is not a subfield of another field can be given either the level 1 or the level 77. If it has level 1, it is treated as a record (see page 42). If it has level 77, it cannot have any subfields.

3 The initial value of a Working Storage field can be defined by means of the VALUE clause (see page 62). If the initial value is not defined in this way, it will be unpredictable.

4 It is conventional but not obligatory to write any data description entries with level 77 first in the section.

CONSTANT SECTION

The Constant Section is used to define constants.Both elementary and group fields whose values are defined by the VALUE clause (see page 62) are defined by data description entries. The section has the following structure:

CONSTANT SECTION.

data description entry.

data description entry.

.

.

.

The following points should be noted about the use of the section:

1 Each field defined must have a unique name unless it is a subfield of another field.

2 A field which is not a subfield of another field can be given either the level 1 or the level 77. If it has level 1, it is treated as a record (see page 42). If it has level 77, it cannot have any subfields.

3 If a field is defined in the Constant Section and its value is not defined either by a VALUE clause in the sentence in which it is named, or in a sentence naming a field of which it is a subfield, or by some other legitimate means such as REDEFINES (see page 54), its value is unpredictable when the program is run.

4 It is conventional but not obligatory to write any data description entries with level 77 first in the section.

LINKAGE SECTION

The Linkage Section may be present only in subroutines. It is used to give descriptions of the subroutine parameters. The section has the following structure:

LINKAGE SECTION.
data description entry
data description entry
.
.
.

The following points should be noted about the use of the section:

1 Each field defined must have a unique name unless it is a subfield of another field.

2 A field which is not a subfield of another field can be given either the level 1 or the level 77. If it has level 1, it is treated as a record (see page 42). If it has level 77, it cannot have any subfields.

3 The VALUE clause may not appear in the Linkage Section, except in condition-name entries.

4 The number and order of record (01 or 77 level) entries in the Linkage Section correspond to the number and order of parameters in the call of the subroutine, unless there is a USING clause in the Procedure Division header for the subroutine (see page 65). The exception to this is the special case mentioned in note 5 below.

5 Records in the Linkage Section may redefine other records but in this case only one parameter in the subroutine call corresponds to both. The redefining parameter merely gives the address of a storage area and the corresponding Linkage Section record gives a description of this storage area. Two Linkage Section records, one of which redefines another, give different descriptions of the same storage area, defined by their corresponding common parameter.

6 COBOL subroutines must contain a Linkage Section which must immediately precede the Procedure Division; conversely, COBOL programs which are not subroutines must not contain a Linkage Section.

DATA DESCRIPTION SKELETON

Record description

FUNCTION A data description entry specifies the characteristics of a particular item of data.

FORMAT 1

```
level-number data-name-1 COPY library name [REPLACING word-1 BY
{word-2      }[                          {word-4      }]]
{identifier-2}[ word-3 BY                {identifier-4}]] ... ].
```

FORMAT 2

```
level number {data-name-1} [REDEFINES data-name-2 ]
             {FILLER     }
[SIZE IS {CHARACTERS}]
         {DIGITS    }
[{PICTURE} IS character-string]
 {PIC    }
[USAGE IS{DISPLAY                }]
         {DISPLAY-1              }
         {DISPLAY-2              }
         {DISPLAY-3              }
         {DISPLAY-4              }
         {DISPLAY-5              }
         {{COMPUTATIONAL  }      }
         {{COMP           }      }
         {{COMPUTATIONAL-1}      }
         {{COMP-1         }      }
[OCCURS integer TIMES[DEPENDING ON data-name-3 ]]
[SIGNED]
[{SYNCHRONIZED}{LEFT }]]
 {SYNC        }{RIGHT}
[POINT LOCATION IS {LEFT } integer {PLACES}]
                   {RIGHT}         {BITS  }
[CLASS IS {ALPHABETIC  }]
          {NUMERIC     }
          {ALPHANUMERIC}
          {AN          }
[JUSTIFIED RIGHT]
[{ZERO SUPPRESS                }                    [LEAVING integer PLACES]][BLANK WHEN ZERO]
 {{CHECK   }       PROTECT    }
 {{CHEQUE  }                  }
 {FLOAT CURRENCY SIGN         }
[VALUE IS literal].
```

FORMAT 3 66 data-name-1 RENAMES data-name-2 [THRU data-name-3]

FORMAT 4 88 condition-name { VALUE IS / VALUES ARE } literal-1
[THRU literal-2 [literal-3 [THRU literal-4]] ...

NOTES

1 A Record Description consists of one or more sentences which describe the characteristics of a particular record. There is a sentence for each name given to a record or one of its parts and a sentence for each condition-name declared in connection with the record. Each sentence causes the next area of data storage either to be given the name which is the data-name in the format or to be allocated without a name if FILLER is written.

2 The *level number* shows the position of an elementary or group item in the organization of a logical record. For purposes of data reference it is necessary to specify subdivisions of a record and to name these subdivisions. The sentence which name the parts follow immediately the sentence which names the whole and are in the same order as the areas of storage which they name. The fact that they are parts of an area already named rather than new areas is indicated by giving them level numbers numerically higher than the level number of the sentence naming the whole. In this way a record can be divided into fields and fields into subfields which can themselves be divided and so on.

Since records are the most inclusive data items, their level numbers start at 1 or 01. Less inclusive data items are assigned higher (not necessarily successive) level numbers up to and including 49. A level number must be an unsigned integer and is written according to the rules for numeric literals.

The level numbers 66, 77 and 88 are used for the following special purposes: 66 is assigned to data-names described by a RENAMES clause for the purpose of regrouping data items; 77 to non-contiguous items in the Working-Storage and Constant Sections which are not subdivisions of other items and are not subdivided; and 88 to entries specifying condition-names associated with particular values of a conditional variable.

3 The size of a field is the sum of the sizes of its elementary subfields, except for those which Redefine or Rename other subfields.

4 Format 1 is used when the COBOL library contains the data description entry. (For COPY see Chapter 8 on page 107.)

5 The clauses may be written in any order with the exception that the REDEFINES clause must immediately follow the data-name.

6 The clauses are described in alphabetical order on pages 44 to 62.

CLASS

FUNCTION The CLASS clause defines the category to which a data item belongs.

FORMAT

```
CLASS IS { ALPHABETIC   }
         { NUMERIC      }
         { ALPHANUMERIC }
         { AN           }
```

NOTES

1 There are three categories of data in COBOL that can be described by the CLASS clause, namely ALPHABETIC, NUMERIC and ALPHANUMERIC. Every elementary item belongs to one of these categories.

2 ALPHABETIC defines a data item consisting of six-bit characters from the set A to Z and space.

3 NUMERIC defines a data item containing a binary, decimal or sterling numeric value.

4 ALPHANUMERIC and AN, which is an acceptable abbreviation for ALPHANUMERIC, define a data item containing characters from the data character set given in Appendix 1.

5 The CLASS clause can be written at any level. If it is written in a sentence describing a group field, it applies to each elementary field in the group and the sentences describing the fields within that group must not contain contradictory statements about the kind of information to be held in them. Alphabetic or numeric fields within an alphanumeric field are not considered contradictory.

6 If the class of an elementary field is not indicated in any way by the COBOL program and cannot be determined from the data description of any group to which it is subordinate, it is assumed to be ALPHANUMERIC.

7 If a field is described by CLASS and PICTURE, the compiler ignores the CLASS clause.

EDITING CLAUSES

FUNCTION The Editing Clauses permit suppression of non-significant zeros and commas, floating currency signs, check protection and the blanking of an item when its value is zero.

FORMAT

```
{ ZERO SUPPRESS                    }
{ { CHECK  }                       }
{ { CHEQUE } PROTECT               } [LEAVING integer PLACES] [BLANK WHEN ZERO].
{ FLOAT CURRENCY SIGN              }
```

NOTES

1 The editing clauses can only be used in sentences describing elementary fields which are numeric and have usage DISPLAY or DISPLAY-n (see USAGE page 59).

2 The editing clauses, with the exception of BLANK WHEN ZERO, are all ways of expression properties about the field which can also always be expressed by using PICTURE (see page 49). They will therefore be explained by reference to PICTURE.

3 If a field is described by the PICTURE clause and an editing clause other than BLANK WHEN ZERO, the editing clause is ignored.

4 ZERO SUPPRESS gives the same information about zero suppression as a Picture containing Zs, If the option LEAVING integer PLACES is not used, the corresponding Picture has one Z for each character of the field to the left of the decimal point which represents a data character. If LEAVING integer PLACES is used, the corresponding Picture has a Z for each data character starting from the left until there are 'integer' more data characters before the decimal point. This is true even when there are unmapped digits before the decimal point. For example, if a field is described as

SIZE 4 POINT RIGHT 1 ZERO SUPPRESS LEAVING 2

the corresponding Picture is ZZZ9P.

The 'integer' is written according to the rules for numeric literals and must be unsigned.

5 CHECK PROTECT gives the same information about cheque protection as a Picture containing asterisks and has the same relationship to such a Picture as ZERO SUPPRESS has to a Picture containing Zs (see pages 45 and 49).

6 FLOAT SIGN gives the same information about floating a currency sign as a Picture containing leading £s and has the same relationship to such a Picture as ZERO SUPPRESS has to a Picture containing Zs (see pages 45 and 49).

7 When the BLANK WHEN ZERO clause is used to describe a destination field, the field will contain nothing but spaces if its value is zero. The clause is ignored by the compiler if it appears in the same sentence as the CHECK PROTECT clause or a PICTURE clause containing the symbol *. BLANK WHEN ZERO may only be written for numeric fields.

FILLER

FUNCTION FILLER is used as an alternative to a data-name in a sentence allows the programmer to map part of the data store without giving it a name.

FORMAT FILLER

NOTES

1 FILLER can be applied to both group and elementary fields.

2 The sentence in which FILLER occurs may contain any of the clauses used to describe data items. It must specify the amount of storage to be allocated but no other information is obligatory.

3 The word FILLER is not a data-name and must not appear in the Procedure Division except in a NOTE statement.

FUNCTION The JUSTIFIED clause specifies non-standard positioning of data within a receiving data item.

FORMAT $\left\{\begin{matrix}\underline{\text{JUSTIFIED}}\\ \underline{\text{JUST}}\end{matrix}\right\}$ $\underline{\text{RIGHT}}$

NOTES

1 The JUSTIFIED clause can be specified only at the elementary item level.

2 The JUSTIFIED clause cannot be specified for an item described as numeric, for a numeric edited data item or for an item whose size is variable.

3 When the receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the left-most characters are truncated. When the receiving data item is described with the JUSTIFIED clause and is larger than the sending data item, the data is aligned at the right-most character position in the data item with space fill.

OCCURS

FUNCTION The OCCURS clause eliminates the need for separate entries for repeated data and supplies information required for the application of subscripts.

FORMAT OCCURS integer TIMES[DEPENDING ON data-name-3]

NOTES

1 The clause causes "integer" contiguous fields to be defined, all of which have the same name and characteristics defined in the sentence in which OCCURS is written. Moreover, if the field described in the sentence in which OCCURS is written has subfields described in subsequent sentences, all "integer" fields have the same subfields.

2 A field and its subfields described by OCCURS is referred to by subscripting (see page 66).

3 The integer is written according to the rules for numeric literals and must be unsigned.

4 The clause must not be used to describe a field whose level number is 01, 66, 77 or 88.

5 A field which has a subfield described as OCCURS integer TIMES includes all occurrences of the subfield.

6 If a field is described as OCCURS integer TIMES, it can be the subfield of a field described as OCCURS integer TIMES. This field in its turn can be the subfield of a field described as OCCURS integer TIMES. The largest field, however, cannot be the subfield of another field described as OCCURS integer TIMES. In other words, tables can have no more than three dimensions.

7 The following notes apply if the DEPENDING option is used.

(a) The DEPENDING option does not mean that the field in whose description it appears is variable in length, but that the number of occurrences of it in a record will vary according to the value contained in the field "data-name-3". The record within which the occurring field is contained will thus be of variable length. "integer" defines the maximum number of occurrences of the field.

(b) Only one such clause may appear in a record, i.e. there must be only one variably occurring field in a record.

(c) The variably occurring field must be either the last elementary field in the record, or a group field containing the last elementary field in the record.

(d) The field whose description contains an OCCURS ...DEPENDING clause must not be contained in a group field whose description contains an OCCURS clause.

(e) "data-name-3" must be a field whose class is defined as numeric and it must have been declared in the Data Division before the declaration of the data field containing the OCCURS ...DEPENDING clause.

(f) "data-name-3" must not be subscripted or qualified.

(g) An OCCURS ...DEPENDING clause must not appear in any record of a file whose FD contains the RECORDING MODE F clause.

(h) The area allocated to an OCCURS ...DEPENDING field in the object program will be equal to the size of the field multiplied by "integer".

(i) In the object program a record with an OCCURS ...DEPENDING clause in its description is considered to have a variable length only when executing input or output statements. In all other operations it is considered to have its fixed maximum length. Similarly in operations on any other group field that contains an OCCURS ...DEPENDING clause in its description the maximum size is assumed.

FUNCTION The PICTURE clause describes the general characteristics and editing requirements for an elementary item.

FORMAT { PICTURE / PIC } IS character-string

NOTES

1 The PICTURE clause can only be used to describe an elementary field.

2 A character string consists of certain allowable combinations of characters in the COBOL character set used as symbols.

3 A Picture must not consist of more than 30 characters.

4 The following characters and combinations of characters can appear in the character string. The notes under the heading Meaning indicate the range of values of the character or bit position when the object program is run and legal operations are performed. In some cases, for example, 0, the information given is not meaningful in isolation and reference should be made to the editing rules in Chapter 10.

Character	Meaning
( )	A left and right parenthesis pair may be used to write a Picture more briefly than would otherwise be the case. The parentheses must enclose one or more contiguous numeric digits. The character immediately preceding the left parenthesis is to be considered to appear as many times as indicated by the number within the parentheses: e.g. A(5) means the same as AAAAA.

The following characters assign characters or bits of storage to the field. The position of the character or bit within the field is indicated by the position of the character within the Picture's character string. For this purpose, Pictures containing parentheses should be considered in their expanded form.

Character	Meaning
A	Represents a character position with contents from the set A to Z and space.
X	Represents a character position with contents from the data character set given in Appendix 1.
9	Represents a decimal digit. See page 60 for the size of a field indicated by a Picture containing '9's.
1	Represents a binary digit. See page 61 for the size of a field indicated by a Picture containing '1's.
S	Indicates that the field contains a signed numeric value. The rest of the Picture must indicate a field which holds an unedited decimal or binary numeric value, i.e. if decimal, all digits are from the set 0 to 9; if binary, all digits are from the set 0 and 1.
Z	Represents a character position with contents from the set 0 to 9 and space.
*	Represents a character position with contents from the set 0 to 9 and *.
£	Represents a character position with contents from the set 0 to 9, space and £.
$	Represents a character position with contents from the set 0 to 9, space and $.
-	Represents a character position with contents from the set 0 to 9, space and -.
+	Represents a character position with contents from the set 0 to 9, space + and -.
CR	These two characters must be next to each other in the order given and must be in the last two positions in the Picture. They represent two character positions which contain either two spaces or CR.

Character	Meaning
DB	D can only be written in the Picture followed immediately by B, in which case, the following rule applies: DB represents two character positions which contain either two spaces or DB. They must be in the last two positions in the Picture.
	Represents a character position which contains a character from the set [,] space [*] £ [+] $ [-]. It can be the last character of a Picture only if it is immediately followed by one of the punctuation symbols [,] [;] [.].
B (except when immediately preceded by D)	Represents a character position which contains a character from the set space [*] £ [+] [-] $ if the field described is numeric, or a space if the field is non-numeric.
0	Represents a character position which contains a character from the set 0, space [*] £ [+] [-] $ if the field described is numeric, or 0 if the field is alphanumeric.
[.]	Represents a character position which contains a character from the set [.] space [*] £ [+] [-] $. It can be the last character of a Picture only if it is immediately followed by one of the punctuation symbols [,] [;] [.].

The following characters give information which affects the meaning of operations performed upon the field. They do not cause storage to be assigned.

Character	Meaning
V	Represents the position to be assumed for the decimal or binary point.
P	Shows that the decimal or binary point does not lie to the immediate left or the immediate right of any digit represented by a character within the Picture. P can appear only either at the left-hand end of the Picture preceded only by 'P's or S, or at the righthand end followed only by 'P's or CR or DB. If a Picture contains n left-hand 'P's, the first digit of the field represents units of $10^{-(n+1)}$ if it is a decimal field or $2^{-(n+1)}$ if it is a binary field. If a Picture contains n right-hand 'P's, the last digit of the field represents units of 10^n if it is a decimal or 2^n if it is a binary field.

If neither V or P appears in a Picture which represents an edited or unedited numeric field, the numeric part of the field is an integer. A right-hand V can be used to indicate an integer but it is unnecessary.

5 The table given below shows the combinations of characters which are valid and invalid Pictures. The effect of combining characters in a Picture can be found in the following way:

(a) Find the character which is to precede in the left-hand column of the table.

(b) Find the character which is to follow in the top row of the table.

(c) The entry in the table for the co-ordinates obtained in this way gives the information required.

The information given applies not only if the two characters are adjacent but also if they appear in the same Picture. For example, once A has been written in the Picture any subsequent 1, S or Z causes the Picture to be invalid.

The two pairs of characters, CR and DB, are treated as one character for the purpose of the table. The characters [(] and [)] do not appear in the table since the validity of a Picture in which they appear is established by expanding the Picture.

The codes in the table have the following meanings:

Code	*Meaning*
blank square	The combination is invalid.
✓	The combination is valid.
X	The combination is valid but the effect is as if the Picture were re-written with all '9's and 'A's replaced by 'X's.
R	The combination is valid provided the second of the two characters being considered is the last character of the Picture except possibly for a P.

9 The combination is valid provided the character 9 does not occur in the Picture anywhere following the second of the two characters being considered.

1 The combination is valid provided the first of the two characters being considered only occurs once.

See also the comments on the position of P given on page 50 .

Following character

Preceding character	A	X	9	1	S	Z	*	£	$	-	+	CR	DB	,	B	0	.	V	P
A	✓	X	X												✓				
X	X	✓	X												✓	✓			
9	X	X	✓							R	R	R	R	✓	✓	✓	✓	✓	✓
1				✓														✓	✓
S			✓	✓														✓	✓
Z			✓			✓				R	R	R	R	✓	✓	✓	✓	✓	✓
*			✓				✓			R	R	R	R	✓	✓	✓	✓	✓	✓
£			✓			1	1	✓		✓	✓	R	R	✓	✓	✓	✓	✓	✓
$			✓			1	1		✓	✓	✓	R	R	✓	✓	✓	✓	✓	✓
-			✓			1	1	1	1	✓				✓	✓	✓	✓	✓	✓
+			✓			1	1	1	1		✓			✓	✓	✓	✓	✓	✓
CR																			
DB																			
,			✓			✓	✓	✓	✓	✓	✓	R	R	✓	✓	✓	✓	✓	✓
B	✓	✓	✓			✓	✓	✓	✓	✓	✓	R	R	✓	✓	✓	✓	✓	✓
0	X	✓	✓			✓	✓	✓	✓	✓	✓	R	R	✓	✓	✓	✓	✓	✓
.			✓			✓	✓	✓	✓	✓	✓	R	R	✓	✓	✓	✓	✓	✓
V			✓	✓		9	9	9	9	9	9	R	R	✓	✓	✓	✓	✓	
P			✓	✓		✓	✓	✓	✓	✓	✓	R	R	✓	✓	✓	✓		✓

If either of the symbols + or - appears after Z, * , £ or $, it must be the only + or - in the Picture.

6 A Picture must contain either at least one of the following characters

A X 9 * Z 1

or at least two of the following characters

+ - £ $

7 The class of a field described by a Picture is implicit in the Picture and the CLASS clause is therefore ignored as unnecessary if written in the same sentence. The class of a field is deduced from its Picture according to the following rules:

(a) The field is alphanumeric if the Picture contains X or any combination of characters (e.g. A and 9) which causes X to be assumed.

(b) The field is alphabetic if its Picture contains A but not X, 0 or 9.

(c) A field is numeric if its Picture does not contain A or X.

8 If the CLASS clause is used to describe a field which contains subfields, the class indicated by the Picture of a subfield must not contradict the class of the group field. However, alphabetic and numeric fields within an alphanumeric field are not contradictory.

9 Numeric fields are classified according to usage which can be indicated by a separate USAGE clause (see page 59). In some cases the USAGE clause must be used but in others a field's usage can be deduced by the compiler from the field's Picture as follows:

(a) The character 1 in a Picture indicates COMPUTATIONAL-1 usage.

(b) The character 9 in the Picture of a field indicates DISPLAY usage if the USAGE clause is not used to describe the field or a group field containing the field.

10 If the Picture of a field indicates a particular usage, a USAGE clause which contradicts the information in the Picture must not be used to describe the field.

11 The usage indicated for a subfield by a Picture must not contradict the usage specified by the USAGE clause for the field containing the subfield.

12 An alphanumeric field is said to be edited if its Picture contains the symbols B or 0.

13 An alphabetic field is said to be edited if its Picture contains the symbol B.

14 A numeric field is said to be edited if its Picture contains any of the symbols Z [*] [-] [+] CR DB [,] 0 [.] £ B or $ or if it is described by any of the clauses ZERO SUPPRESS, CHECK PROTECT, FLOAT SIGN or BLANK WHEN ZERO.

15 The effect of differences of class, editing and usage on operations specified in the Procedure Division is described in Chapter 10.

POINT LOCATION

FUNCTION The POINT LOCATION clause defines an assumed decimal or binary point in relation to the right-hand end of the field.

FORMAT

POINT LOCATION IS { LEFT / RIGHT } integer { PLACES / BITS }

NOTES

1 The POINT LOCATION clause must only be used to describe an elementary numeric field.

2 The integer must be written according to the rules for numeric literals and must represent an unsigned integer.

3 The word BITS specifies that the distance of the point from the least significant position of the field is measured in binary digits. If it is omitted or PLACES written the distance is measured in decimal digits. BITS must be written only if the field has usage COMPUTATIONAL-1.

4 If a field described by the POINT clause is also described by the PICTURE clause, the compiler prints the POINT clause but does not use it.

REDEFINES

FUNCTION The REDEFINES clause allows the same computer storage area to contain different data items.

FORMAT level-number data-name-1 REDEFINES data-name-2

NOTES

1 This clause specifies that the field being defined by the sentence starts at the same storage position as data-name-2.

2 The level numbers of data-name-1 and data-name-2 must be identical, but must not be 66, 77 or 88.

3 Data-name-2 must be the last previous field defined with the level number of data-name-1. Definitions of fields with numerically less level numbers must not intervene between the definitions of data-name-2 and data-name-1.

4 The level number of data-name-2 must not be 01 when the REDEFINES clause is used in the File Section. It can be 01, however, if the clause is used in the Working-Storage, Constant or Linkage Section.

5 If data-name-2 is defined in a sentence which contains the OCCURS clause, the area occupied by all occurrences of data-name-2 is redefined.

6 Neither data-name-1 nor any of its subfields may be described by the VALUE clause. The VALUE clause can be used, however, to describe condition names related to data-name-1 or its subfields.

7 Unless the area being redefined is an 01 level in the Working-Storage Section, the redefining area must occupy the same amount of store as the redefined area.

RENAMES

FUNCTION The RENAMES clause permits alternative, possible overlapping, groupings of elementary items.

FORMAT 66 data-name-1 RENAMES data-name-2 [THRU data-name-3] .

NOTES

1 A sentence containing level number 66 and the RENAMES clause has two uses:

(a) It can be used without the THRU option to give another name to an already defined elementary or group field. In this case data-name-1 has the same meaning and attributes as data-name-2.

(b) It can be used with the THRU option to define a group field called data-name-1 consisting of all the fields from data-name-2 to data-name-3 inclusive. Group fields can thus overlap. For example, if data-name-2 is the second subfield of a group field, overlapping occurs. Data-name-2 and data-name-3 can both be either elementary or group fields.

2 A sentence containing level number 66 and RENAMES can only be followed in the same record description by other sentences of the same kind.

3 Data-name-2 and data-name-3 must be names of fields already described in the same record description. Data-name-3, if mentioned, must follow data-name-2 and must not be one of its subfields. Neither may be defined with level numbers 01, 66, 77 or 88

4 Data-name-1 may not be used as a qualifier and can only be qualified by the name of the record or file within which it is defined.

5 Neither data-name-2 nor data-name-3 may be defined in a sentence in which the OCCURS clause is used, nor may either of them be a subfield of a field described by the OCCURS clause.

SIGNED

FUNCTION The SIGNED clause specifies the operational sign of an elementary item.

FORMAT SIGNED

NOTES

1 The SIGNED clause indicates the use of a standard operational sign. When the clause appears in an entry with the Picture clause, it is ignored by the compiler and printed for documentation purposes only.

2 SIGNED has the same meaning as S in the Picture clause. It can only be used to describe a field for which it would be possible to write a Picture containing S (see page 49).

3 SIGNED is sufficient to indicate that the associated field is numeric.

FUNCTION The SIZE clause is used to specify the number of characters in a data item.

FORMAT

SIZE IS integer { CHARACTERS / DIGITS }

NOTES

1 The size of an elementary field must be specified by a SIZE or PICTURE clause in the sentence which defines the field. If both are used in the same sentence, The Picture is used to determine the size.

2 The integer in the SIZE clause must be a numeric literal which represents an unsigned integer.

3 The size of a field described by SIZE may vary according to the class and usage of the field. If the field is numeric with usage COMPUTATIONAL, it occupies a number of "integer" decimal digits not counting a possible sign. If the field is numeric with usage COMPUTATIONAL-1, it occupies "integer" bits of 1900 Series storage not counting a possible sign. If the field is of any other class or usage, it occupies "integer" characters of 1900 Series storage not counting a space needed for a possible sign which is specified in all cases by SIGNED or S in the Picture of the field.

4 The storage space occupied by COMPUTATIONAL and COMPUTATIONAL-1 fields for particular specified sizes is given on page 60.

5 If the SIZE clause is used to describe a group field, it is considered to be for documentation purposes only and is ignored by the compiler.

SYNCHRONIZED

FUNCTION The SYNCHRONIZED clause specifies the positioning of an elementary item within a computer word.

FORMAT

$$\left\{\begin{matrix}\underline{\text{SYNCHRONIZED}}\\ \underline{\text{SYNC}}\end{matrix}\right\}\left\{\begin{matrix}\underline{\text{LEFT}}\\ \underline{\text{RIGHT}}\end{matrix}\right\}$$

NOTES

1 The SYNCHRONIZED clause specifies that the field so described is to occupy the minimum possible number of 1900 series words and that no other field will occupy the same words.

2 The clause must only be used to describe elementary fields.

3 If the option SYNCHRONIZED LEFT is used, any unused storage in the words allocated to the field is in the least significant part of the last word. If the option SYNCHRONIZED RIGHT is used, any unused storage is in the most significant part of the first word.

4 A group field contains the unused parts of words assigned to its synchronized subfields. For example, a group field whose first elementary subfield is SYNCHRONIZED RIGHT and does not occupy all of the first 1900 series word it is assigned, starts at the beginning of that 1900 series word.

5 If a field is synchronized, the Data Division should be written so that the previous field ends at the end of a 1900 Series word. If necessary, this can be done by declaring a FILLER. If it is not done, the compiler will print a cautionary message and supply the FILLER.

6 All the occurrences of a synchronized field described by the OCCURS clause or of a synchronized subfield of a field described by the OCCURS clause are synchronized.

7 SYNCHRONIZED can be written as SYNC.

8 If a record in an input field contains fields that are described as SYNCHRONIZED, this does not imply any distribution of information into words when the record is read by the object program. Information on an external medium can always be considered as an exact image of the same information inside the computer store.

USAGE

FUNCTION The USAGE clause specifies the format of a data item in the computer storage.

FORMAT

```
USAGE IS { DISPLAY
         | DISPLAY-1
         | DISPLAY-2
         | DISPLAY-3
         | DISPLAY-4
         | DISPLAY-5
         | { COMPUTATIONAL
           | COMP
         | { COMPUTATIONAL-1
           | COMP-1 }
```

NOTES

1 The USAGE clause is used to specify to the compiler the dominant use of a field. The compiler uses it to decide the way information is to be assumed to be recorded in the field.

2 The clause can be written at any level. If it is used to describe a group field, it applies to every elementary field within the group field. The usage stated for a field must not contradict the usage of the group field of which it forms a part.

3 If the usage of an elementary field or the group field of which it forms a part is not stated, it is assumed to be COMPUTATIONAL-1 in the case of a field whose Picture contains the character 1 and DISPLAY in all other cases.

4 The usage DISPLAY specifies that each character or decimal digit is represented by a 1900 six-bit character. Operational signs are represented by special coding ("overpunching") of the leftmost character. If a field is negative, the numbers 1 to 9 in the leftmost character position appear as J to R respectively and a leading zero appears as either – (minus) or " (quotes) depending on whether leading zeros are represented by spaces or zeros respectively. Leading zeros are usually represented by zeros but may also be represented by spaces when data is input from punched cards in which case leading zeros are not punched. If the leftmost character is not "overpunched" to indicate a negative value, the value is positive. Whenever an object program converts to DISPLAY form, leading zeros are represented by zeros unless explicitly suppressed by editing.

5 The usage DISPLAY-1 specifies a sterling field in which each pound digit, the tens of shillings digit, the units of shillings digit, the pence digit (radix 12) and each decimal place of pence is represented by a 1900 six-bit character. Operational signs and leading zeros are dealt with in the same way as for DISPLAY with the addition that leading zeros within the shillings field can also be represented by spaces.

Within the 1900 series, 10d and 11d are normally represented as single characters by : (colon) and ; (semicolon) respectively. However, on cards, 10d and 11d are often represented by holes punched in the 10 and 11 rows which are read in as & (ampersand) and - (minus) respectively. Either form is acceptable as input, but whenever a COBOL object program converts to DISPLAY-1, 10d and 11d will always be represented by a colon and semicolon respectively.

6 The usage DISPLAY-2 specifies a sterling amount similar to DISPLAY-1 except that the pence value is represented by two six-bit characters, the first representing tens of pence and the second the units of pence. Operational signs and leading zeros are as for DISPLAY. The leading zeros within the shillings field and the pence field can be represented by spaces, e.g. £2/6/9 input to a DISPLAY-2 field with PICTURE 9(7) as ▽▽2▽6▽9 is taken to represent 0020609 for the purposes of arithmetic. (▽ denotes one space.)

7 The usage DISPLAY-3 is equivalent to DISPLAY with the exception that if a DISPLAY-3 field contains an operational sign indicated by an S in the PICTURE the sign is held as an additional 1900 character preceding those of the field.

This character is + (plus) or ▽ (space) for positive values and - (minus) for negative values.

8 The usage DISPLAY-4 is equivalent to DISPLAY-1 except that as with DISPLAY-3 the operational sign is held as a 1900 character.

9 The usage DISPLAY-5 is equivalent to DISPLAY-2 with the same exception as DISPLAY-3 and DISPLAY-4.

10 In signed DISPLAY, DISPLAY-1 and DISPLAY-2 fields the operational character S is not a mappable character, whilst in DISPLAY-3, DISPLAY-4 and DISPLAY-5 fields it is. For example, a three digit numeric field, whether signed by an overpunch on the most significant digit or by an additional preceding sign character, has a Picture of S999.

11 The differences between the various types of DISPLAY fields are summarized in the table below.

	Operational sign held as		Pence of sterling field held as	
	'Overpunch'	Additional character	1 character	2 characters
DISPLAY	√			
DISPLAY-1	√		√	
DISPLAY-2	√			√
DISPLAY-3		√		
DISPLAY-4		√	√	
DISPLAY-5		√		√

12 A field described as usage COMPUTATIONAL must be numeric and the compiler assumes it to be so if it is not explicitly stated. The clause specifies that a number is held in the form best suited to the most efficient execution of arithmetic on it. The rules concerning COMPUTATIONAL fields are exactly the same as for DISPLAY fields with the exceptions that they cannot be described as edited and that they require different storage space. Effectively arithmetic on such fields is decimal and no binary truncation errors occur.

13 The amount of internal storage occupied by a COMPUTATIONAL field is given in the table below:

Decimal digits specified by the PICTURE	Storage when the PICTURE does not specify a sign (number of bits)	Storage when the PICTURE specifies a sign (number of bits)	Storage for SYNCHRONIZED fields (number of words)
1	4	5	1
2	7	8	1
3	10	11	1
4	14	15	1
5	17	18	1
6	20	21	1
7	25	26	2
8	28	29	2
9	31	32	2
10	35	36	2
11	38	39	2
12	41	42	2

14 The usage COMPUTATIONAL-1 specifies that a number is held in binary form and that each binary digit is represented by a "1" in the Picture of the field. The field must be numeric and the compiler assumes it to be so if it is not explicitly stated. Such a field occupies the number of bits indicated in the Picture or SIZE clause plus one if an operational sign is declared. It must not, however, occupy more than 48 bits of 1900 Series storage. The maximum value which can be held in a field described as COMPUTATIONAL-1 and n bits in length, excluding the sign, is $2^n - 1$ if n is less than 24, $2^{n-1}-1$ if n is greater than or equal to 24 and less than 48 and $2^{n-2}-1$ if n is equal to 48. In other words the 24th and 48th bits of a field may be used to hold an operational sign but will not be used if they do not.

Similar remarks apply to the point position of fields.

For example, a field with the Picture 1(3)V1(24) has a point position of 23 left i.e. 23 binary digits after the point. For the purposes of arithmetic this is equivalent to a field with the Picture 1(4)V1(23) since the properties of the field are the same whether the 'unused' bit (the 24th from the right) is described as being before or after the point. Both fields have three integral binary places and 23 places after the point.

15 COMPUTATIONAL and COMPUTATIONAL-1 can be written as COMP and COMP-1 respectively.

VALUE

FUNCTION The VALUE clause defines the value of constants, the initial value of working-storage items or the values associated with a condition-name.

FORMAT 1 VALUE IS literal

FORMAT 2 { VALUE IS / VALUES ARE } literal-1 [THRU literal-2] [literal-3 [THRU literal-4]] ...

NOTES

1 If the VALUE clause is used to describe a field in the Constant Section, format 1 must be used and the literal is the value to be held in the field throughout the running of the program. It is the programmer's responsibility to ensure that the value of the field is not changed by the program.

2 If the VALUE clause is used to describe a field in the Working-Storage Section, format 1 must be used and the literal is the value the field will contain at the start of the program.

3 If the VALUE clause appears in a sentence defining a field in the File Section, it is printed by the compiler but has no other effect, since the values of all fields defined in the File Section are undefined at the start of the object program. An error message is printed but is ignored.

4 Format 2 may only be used to describe a condition name associated with a field in one of the three sections of the Data Division (see page 63).

5 A figurative constant can be substituted for a literal in either of the formats.

6 The clause must not appear in the same sentence as OCCURS or REDEFINES, a sentence defining a subfield of a field described by OCCURS or REDEFINES, or a sentence defining an edited field.

7 If the clause is used to describe a group field, it must not be used to describe any of its subfields.

8 Subject to the above restrictions, a VALUE clause describing a field which is not a condition-name is considered to have the same effect as a Procedure Division statement " MOVE literal TO data-name" executed before the Procedure Division itself, data-name being the field described by the VALUE clause.

9 The rules for moving literals to various types of fields are given in Chapter 10. As explained there, the values of sterling fields can be written as pence or as pounds, shillings and pence and the values of binary (i.e. COMPUTATIONAL) fields can be written in binary or decimal form.

CONDITION NAMES ASSOCIATED WITH DATA FIELDS

A condition-name is the name assigned to a specific set of values or range of values within the complete set of values that a data item may assume. The data item itself is called a conditional variable. The condition-name is devised by the programmer according to the rules for devising data-name given on page 60 and is defined by a sentence with the form:

88 condition-name VALUE-clause.

The sentence defining the condition-name can be written in any of the three sections of the Data Division. It must follow the sentence defining the field with which it is associated and may be separated from it only by sentences defining condition-names associated with the same field.

A condition-name can be used as a condition (see page 70) in the Procedure Division with the value "true" or "false". Its meaning is obtained by using the literals in the VALUE clause to compose a condition with the data name with which the condition-name is associated. Thus

VALUE IS literal

is equivalent to

data-name EQUALS literal

and every subsequent "literal" is equivalent to

OR data-name EQUALS literal

Similarly

VALUE IS literal-1 THRU literal-2

is equivalent to

data-name NOT LESS THAN literal-1 AND

data-name NOT GREATER THAN literal-2

and every subsequent "literal THRU literal" is equivalent to

OR data-name NOT LESS THAN literal AND

data-name NOT GREATER THAN literal.

For example, if the Data Division contained the following two sentences

02 A PICTURE 999.

88 CORRECT VALUE 4, 10 THRU 18.

CORRECT may be used in the Procedure Division to mean

A EQUALS 4 OR A NOT LESS THAN 10 AND A NOT GREATER THAN 18.

Condition-names may also be associated with fields which contain the OCCURS clause and consequently may be subscripted. For example, if the Data Division contained the following two sentences:

02 DAY PICTURE 999 OCCURS 7 TIMES.

88 HOLS VALUE 1 THRU 8.

the condition

IF HOLS (5).....

would be true if the value of the 5th DAY is in the range 1 to 8.

ALLOCATION OF STORAGE TO DATA

The 1900 Series data store is divided into words, each of which contains four six-bit characters, i.e. 24 bits per word.

If a COBOL program contains only character fields, the layout of the object program data store is extremely simple. The first elementary field is allocated the required number of characters, starting at the first character of the first word assigned to data storage in the object program. The first character of the second elementary field is assigned to the character immediately following that

assigned to the last character of the first field. The only occasions when this rule is broken are on meeting a new record, when the first field is assigned space starting at the beginning of the next clear word, and on meeting a REDEFINES or RENAMES clause.

If the COBOL program contains bit fields (i.e. COMPUTATIONAL or COMPUTATIONAL-1 fields) or fields described as SYNCHRONIZED, the layout of the object program data store is more complicated. Bit fields, unless they start a record or appear with a REDEFINES or RENAMES clause, are allocated storage starting at the next available bit within the data store. Thus, bit fields unlike character fields do not necessarily begin and end on character boundaries. This may cause gaps to occur between data areas within a record. Consider, for example, a DISPLAY field followed by a COMPUTATIONAL or COMPUTATIONAL-1 field requiring 38 bits of storage. The bit field will be allocated the next 38 bits of storage after the last character of the DISPLAY field. Although the bit field in this example begins on a character boundary (as it follows a character field) it ends on the second bit within a character. If the next field is a bit field, it is allocated storage starting at the next bit. However, if it is a DISPLAY field it cannot immediately follow the second field as it would not then start on a character boundary. A FILLER of four bits has therefore to be included after the bit field in this example to enable the DISPLAY field to start on a character boundary. If the programmer omits such a FILLER, it is inserted for him by the compiler.

Synchronization

The clauses SYNCHRONIZED LEFT and SYNCHRONIZED RIGHT can sometimes be used to enable the compiler to take advantage of the fact that the object machine is a word machine and so produce better code. If a field is described as SYNCHRONIZED LEFT, it is allocated as many consecutive whole words of storage as are necessary to hold it and is left-justified within these words. SYNCHRONIZED RIGHT has the same effect except that the field is right-justified.

Thus, for example, on the 1900 Series (four characters per word) a five-character field described as SYNCHRONIZED LEFT is allocated two consecutive whole words and is stored in the first five characters. Although the field takes two words of storage, its size is still only five characters when referenced in the program. The contents of the remaining three characters are undefined.

On the 1900 Series there is nothing to be gained by describing character fields as SYNCHRONIZED because there are adequate character manipulating orders in the 1900 Series object code. However, describing a COMPUTATIONAL or COMPUTATIONAL-1 field as SYNCHRONIZED RIGHT means that it is held in as many whole words as required and is right-justified within them and that, if it is signed, the sign bit is propagated throughout the leading 'excess' bits in the first word. If it is not signed, these leading bits are zero. This means that the field is stored in the form most convenient for fast arithmetic on the 1900 Series.

A COMPUTATIONAL or COMPUTATIONAL-1 field described as SYNCHRONIZED LEFT means that the field is left-justified within the words assigned, the 'excess' bits being undefined.

Since a SYNCHRONIZED field always starts on a word boundary the previous field within the record must end on a word boundary. FILLER can be used to achieve this. If the programmer omits such a FILLER, it is inserted by the compiler.

If a record in an input field contains fields that are described as SYNCHRONIZED, this does not imply any distribution of information into words when the record is read by the object program. Information on an external medium can always be considered as an exact image of the same information inside the computer store.

Chapter 6 Procedure division

STRUCTURE

The Procedure Division consists of the division heading followed by one or more sections or one or more paragraphs containing procedures.

Heading format

PROCEDURE DIVISION [USING identifier-1...]

The USING clause may be present only in COBOL subroutines, where even then it is not obligatory. When present it indicates the number and order of the parameters required by the subroutine and must correspond to the number and order of parameters in the CALL of the subroutine. Identifier-1... must be the names of records defined in the Linkage Section of the subroutine. Not all Linkage Section records need be named in the clause, nor need the order of identifiers correspond to the order of records in the Linkage Section.

Procedures

A procedure is composed of a paragraph, or a group of successive paragraphs, or a section, or a group of successive sections within the Procedure Division. If one paragraph is in a section, all paragraphs must be in sections. A procedure-name is either a paragraph-name or a section-name.

The end of the Procedure Division and the physical end of the program is that physical position in a COBOL source program after which no further procedures appear. (An end marker record of four asterisks in columns 1-4 follows the last procedure statement. This informs the compiler that the end of the source program has been reached.)

A section consists of a section header followed by one or more successive paragraphs. A section header has the following format:

section-name SECTION [integer]

A section ends immediately before the next section-name or at the end of the Procedure Division.

A paragraph consists of a paragraph-name followed by one or more successive sentences. It ends immediately before the next paragraph-name or section-name or at the end of the Procedure Division.

A sentence consists of one or more statements and is terminated by a full stop followed by a space. There are three types of sentences: imperative, compiler directing and conditional sentences.

A statement is a syntactically valid combination of words and symbols beginning with a COBOL verb. There are three types of statements: imperative, compiler directing and conditional statements.

To enhance the readability of a sentence or statement, statements can be separated by the word THEN, or a semi-colon or a comma. These separators must not be followed immediately by another separator.

Imperative statements

An imperative statement indicates a specific action to be taken by the object program. It can consist of a sequence of imperative statements each possibly separated from the next by a separator. The imperative verbs are:

ACCEPT	DIVIDE	OPEN	SORT
ADD	EXAMINE	PERFORM	STOP
ALTER	EXIT	RELEASE	SUBTRACT
CLOSE	GO	RETURN	WRITE
DELETE	MOVE	REWRITE	
DISPLAY	MULTIPLY	SEEK	

Compiler directing statements

A compiler directing statement directs the compiler to perform specified actions during the compilation of the object-program. It consists of one of the compiler directing verbs, CALL, COPY, ENTER and NOTE, and its operands.

Conditional statements

A conditional statement specifies that the truth or falsity of a condition is to be established and that the subsequent action of the program depends upon this. It is one of the following statements either preceded or not by an imperative statement:

a statement beginning IF

a statement beginning READ

a statement beginning ADD and having a SIZE ERROR option specified

a statement beginning SUBTRACT and having a SIZE ERROR option specified

a statement beginning MULTIPLY and having a SIZE ERROR option specified

a statement beginning DIVIDE and having a SIZE ERROR option specified

a statement beginning WRITE and having an INVALID KEY option specified

a statement beginning DELETE and having an INVALID KEY option specified

a statement beginning PERFORM and having an UNTIL option specified

ELEMENTS USED IN THE PROCEDURE DIVISION

Data references

The Procedure Division statements may refer to data by using any of the following items:

LITERALS

1 Non-numeric literals (see page 15).

2 Numeric literals (see page 15).

3 Figurative constants (see page 16).

DATA NAMES

See page 15 for the rules for inventing data names and pages 33 and 34 for the meaning of data names.

QUALIFIED DATA NAMES

See page 17 for the form of writing a qualified data name. A file name cannot be qualified nor can a name declared in the Working-Storage, Constant or Linkage Section with level 01 or 77.

SUBSCRIPTED DATA NAMES

These have the format:

data-name (subscripts)

The punctuation rules when using left and right parentheses are explained on page 14. "Subscripts" in the format consist of one, two or three data references each of which can be a numeric literal representing a positive integer, the data name of a field defined as containing an integer or such a data name qualified. A data name used as a subscript cannot be subscripted.

A data name must be subscripted if it or a field of which it is a subfield is defined in a sentence in the Data Division which contains the OCCURS clause (see page 48). If the field which is described by OCCURS is a subfield of another field also described by OCCURS, two data references are necessary as the subscripts. If that field in its turn is a subfield of a field described by OCCURS, three data references are necessary as the subscripts. The first subscript refers to the OCCURS of the field with the lowest level number and additional subscripts refer to OCCURS clauses describing successively more enclosed fields. For example, if an extract from a Data Division is as follows:

```
01 REGION.
  02 COUNTY OCCURS 10 TIMES.
    03 DISTRICT OCCURS 20 TIMES.
      04 SECTOR OCCURS 5 TIMES.
        05 PARTA PICTURE XX.
        05 PARTB PICTURE XXXXX.
```

COUNTY requires one subscript; DISTRICT requires two subscripts the first of which defines the COUNTY and the second the DISTRICT within the COUNTY; SECTOR, PARTA and PARTB all require three subscripts the first of which defines the COUNTY, the second the DISTRICT and the third the SECTOR.

If the value of a subscript is n, it refers to the nth occurrence of the field it distinguishes, where n is a positive integer. For example, the subscript of the first occurrence of a field is 1 and the fifth occurrence is 5. The value of a subscript must not be greater than the number in the relevant OCCURS clause.

The following are examples of correctly subscripted data names, assuming that the names involved are correctly defined in the Data Division.

```
A (1)
A (B)
A (B, 1, C)
A OF X (B, 1, C)
A OF X (B OF Y, E, C OF G OF H)
```

The commas inserted between the subscripts are required according to the rules of COBOL and should be inserted for compatibility with other compilers. 1900 COBOL allows the subscripts to be separated also by spaces.

The qualifiers are considered as part of the data-name and thus precede any subscripts. That is, the last two examples must not be written as

```
A (B, 1, C) OF X
A (B OF Y, E, C OF G OF H) OF X
```

The CORRESPONDING option

In the formats of the ADD, SUBTRACT and MOVE statements containing the CORRESPONDING option, identifier-1 and identifier-2 must refer to group items which are referred to in the following description as d_1 and d_2.

A pair of data items, one from d_1 and one from d_2, correspond if all these conditions exist:

1 Neither d_1 nor d_2 is a data item with level number 66, 77 or 88.

2 Each of the pair of data items has the same name and the same qualification up to, but not including, d_1 and d_2.

3 At least one of the data items is defined as elementary in the case of MOVE CORRESPONDING, and both of them must be elementary *and* numeric in the case of ADD CORRESPONDING and SUBTRACT CORRESPONDING.

4 All data items subordinate to d_1 and d_2 with level number 66 or 88 are ignored.

5 *a* data item subordinate to d_1 or d_2 and containing a REDEFINES or occurs clause is ignored. However, d_1 and d_2 *may* have REDEFINES or occurs clauses or be subordinate to data items with such clauses.

For example, if two records are defined as:

```
01      OLD-REC.
  02        AREA-A.
    03          SUB-AREA-A1 PIC 999.
```

```
           03    SUB-AREA-A2 PIC 999.
           03    SUB-AREA-A3 PIC 9(5).
           03    SUB-AREA-A4 PIC 1(6) OCCURS 7.
        02    AREA-B.
           03    SUB-B PIC 9(7).
        02    SUB REDEFINES AREA-B PIC X(7).
        02    SWITCH PIC 9.
```

and

```
     01    NEW-REC.
        02    AREA-A.
           03    SUB-AREA-A1 PIC 999.
           03    SUB-AREA-A2 PIC XXX.
           03    SUB-AREA-A3.
              04    SUB-33 PIC 9(2).
              04    SUB-333 PIC 9(3).
           03    SUB-AREA-A4 PIC 1(42).
        02    AREA-BB.
           03    SUB-B PIC 9(4).
        02    AREA-B PIC 9(7).
        02    SWITCH-B PIC 9.
        88    SWITCH VALUE 1.
        02    SUB-D PIC X(2).
```

The corresponding items in OLD-REC and NEW-REC ARE:

1 In the case of MOVE CORRESPONDING:

SUB-AREA-1

SUB-AREA-2

SUB-AREA-3

AREA-B

2 In the case of ADD or SUBTRACT CORRESPONDING:

SUB-AREA-1

SUB-AREA-2 in NEW-REC is alphanumeric and therefore does not correspond.

Conditions

A condition causes the object program to select between alternate paths of control depending upon the truth value of a test. Conditions are used in IF and PERFORM statements.

A condition is one of the following:

relation condition

class condition

condition-name condition

switch-status condition

sign condition

RELATION CONDITIONS

A relation condition causes a comparison of two operands, each of which may be an identifier or a literal. The format for such a condition is as follows:

$$\left\{\begin{array}{l}\text{identifier-1}\\ \text{literal-1}\end{array}\right\} \text{ relational-operator } \left\{\begin{array}{l}\text{identifier-2}\\ \text{literal-2}\end{array}\right\}$$

A literal must not be compared with another literal. The relational operators permitted together with the values of the conditions in different situations is shown in the table below.

Relational operator	Value of condition when		
	1st operand is greater	2nd operand is greater	Operands are equal
IS GREATER THAN EXCEEDS IS >	true	false	false
IS NOT GREATER THAN IS NOT >	false	true	true
IS LESS THAN IS <	false	true	false
IS NOT LESS THAN IS NOT <	true	false	true
IS EQUAL TO EQUALS IS =	false	false	true
IS NOT EQUAL TO IS NOT = IS UNEQUAL TO	true	true	false

The following rules should be applied in all cases to determine whether the first operand is greater than, less than or equal to the second. The rules do not necessarily describe the actions performed by the program to make the comparison.

1 If one of the operands is a literal, it is considered to have been moved before the comparison to a field of the same class, usage and size as that of the field with which it is to be compared. The literal is truncated and/or zero or space filled if necessary. The rules given below for comparing fields then apply.

2 If an alphanumeric or alphabetic field is compared with an alphanumeric or alphabetic field, the contents of the two fields are compared character by character starting with the leftmost. If the character of one field has a value greater than that of the other, the field to which it belongs has a greater value. If the two characters are the same the next character to the right is selected from each field and compared in the same way. If the two fields are not of equal length, the shorter is considered to be extended to the length of the longer for the purpose of comparison and the extra character positions so added are considered to contain spaces. If there are no more characters in either field, the two fields are equal in value. A character has a value greater than another if it appears later in the alphanumeric Data Character Set given in Appendix 1.

3 If an edited numeric field with usage DISPLAY or DISPLAY-n is being compared, it is treated as an alphanumeric field and must conform to the rules for comparing alphanumeric fields.

4 If two unedited numeric fields are compared, the second is considered to be subtracted from the first according to the rules for subtraction given in Chapter 9. The first field is then greater than, less than or equal to the second depending on whether the result is positive, negative or zero respectively.

5 If the usages of the fields compared are as follows, the usual restriction on the size of arthmetic operands does not apply, that is, they can be longer than 12 digits plus the sign.

DISPLAY compared with DISPLAY

DISPLAY compared with DISPLAY-3

DISPLAY-3 compared with DISPLAY-3

DISPLAY-3 compared with DISPLAY

DISPLAY-2 compared with DISPLAY-2

DISPLAY-2 compared with DISPLAY-5

DISPLAY-5 compared with DISPLAY-5

DISPLAY-5 compared with DISPLAY-2

The rules governing the size of operands apply in all other cases. (See Chapter 9.)

6 If an unedited numeric field with usage DISPLAY or DISPLAY-n is compared with an alphanumeric field, the numeric field is assumed to be alphanumeric. Alphanumeric fields must not be compared with numeric fields with usage COMPUTATIONAL or COMPUTATIONAL-1.

7 Alphabetic fields can be compared only with alphabetic or alphanumeric fields.

8 Non-elementary fields may occur as operands in a comparison statement provided that:

(a) they begin on a character boundary and are an integral number of characters in length;

(b) they do not exceed 120 characters in length.

For purpose of comparison such fields are considered to be alphanumeric.

ABBREVIATED RELATION CONDITIONS

When relation conditions are written in a consecutive sequence, any relation condition except the first may be abbreviated by:

1 the omission of the subject of the relation condition.

2 the omission of the subject and relational operator of the relation condition.

Within a sequence of relation conditions both forms of abbreviation may be used. The effect of using such abbreviations is as if the omitted subject was replaced by the last preceding stated subject or the omitted relational operator was replaced by the last preceding stated relational operator.

For example,

1 IF A = B OR > C AND < D

is equivalent to

IF A = B OR A > C AND A < D

2 IF A = B AND C OR D

is equivalent to

IF A = B AND A = C or A = D

3 IF A > B OR = C OR D

is equivalent to

IF A > B OR A = C OR A = D

Ambiguity may result from using NOT in abbreviated conditions, and in such cases NOT is interpreted as a logical operator instead of as part of a relational operator.

Thus:

IF A > B AND NOT > C OR D

is equivalent to

IF A > B AND (NOT A > C) OR A > D

with or without the brackets.

A statement of the type:

IF A = B OR C

will initially be interpreted by the compiler as an abbreviated condition, equivalent to:

IF A = B OR A = C

However if C is in fact a *condition name*, the compiler will print Caution message 219 or 814, and will then reinterpret the statement, correctly, as

IF A = B OR C

instead of as an abbreviated form.

SIGN CONDITIONS

A sign condition determines whether or not the algebraic value of a numeric operand is less than, greater than or equal to zero. The format for such a condition is as follows:

identifier is [NOT] { POSITIVE / NEGATIVE / ZERO }

An operand is positive if its value is greater than zero, negative if its value is less than zero and zero if its value is equal to zero. Expressed differently:

IS [NOT] POSITIVE is equivalent to IS [NOT] GREATER THAN ZERO

IS [NOT] NEGATIVE is equivalent to IS [NOT] LESS THAN ZERO

IS [NOT] ZERO is equivalent to IS [NOT] EQUAL TO ZERO

CLASS CONDITIONS

A class condition determines whether an operand is numeric or alphanumeric. The format for such a condition is as follows:

identifier IS [NOT] { NUMERIC / ALPHABETIC }

The [NOT] NUMERIC test must only be applied to a field which is either alphanumeric, or numeric whose usage is DISPLAY or DISPLAY-n, or to a group field. The [NOT] ALPHABETIC test must only be used with a field which is alphabetic or alphanumeric or with a group field.

If the ALPHABETIC test is used, the value of the field to which the identifier belongs is examined character by character. If all characters are from the set A to Z and space, the value of the condition is true; otherwise, it is false. NOT ALPHABETIC has the opposite meaning. A group field is treated as if it were alphanumeric and can only be tested in this way if it starts on a character boundary and occupies a whole number of characters. The format of the field is ignored when making the test, each 1900 Series character being considered as a character of the field.

If the NUMERIC test is made of a numeric field with usage DISPLAY or DISPLAY-3, the field is examined to see if it contains only characters from the set 0 to 9 together with a possible sign incorporated in the first character if the field is signed. If it does, the value of the condition is true; otherwise, it is false.

NOT NUMERIC has the opposite meaning. An alphanumeric field is examined to see if it contains only characters from the set 0 to 9.

A group field is treated as if it were alphanumeric and can only be tested if it starts on a character boundary and occupies a whole number of characters.

If the NUMERIC test is made of a field with usage DISPLAY-1, DISPLAY-2, DISPLAY-4, DISPLAY-5, the field is examined to see if it contains a legitimate value; for example, in the case of DISPLAY-1 that all characters after the implied decimal point position are in the range 0 to 9, that the character before the point position is in the range 0 to 11, that the two characters before that are in the range 00 to 19, and that all other characters are in the range 0 to 9 except the leftmost which may in addition indicate a sign. If the field contains a legitimate value, the value of the condition is true; otherwise, it is false. NOT NUMERIC has the opposite meaning.

If a field is described as numeric, leading spaces are legitimate representations of leading zeros and do not cause the value to be classified as non-numeric. However, if a field is described as alphanumeric, leading spaces do not represent leading zeros and therefore do not cause the value to be classified as numeric even if the remaining characters of the field are numeric. Moreover, a space is legitimate in the ten shillings position of a sterling field or in the tens of pence position of a sterling field with usage DISPLAY-2 or DISPLAY-5.

CONDITION-NAME CONDITIONS

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name. The format for such a condition is as follows:

[NOT] condition-name

If the condition-name is associated with a range or ranges of values, the conditional variable is tested to determine whether or not its value falls in this range which includes the end values.

The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

SWITCH-STATUS CONDITIONS

A switch-status condition determines the on or off status of a hardware switch named in the Special-Names paragraph of the Environment Division. The format for such a condition is as follows:

[NOT] condition-name

The result of the test is true if the switch is set to the specified position corresponding to the condition-name.

COMPOUND CONDITIONS

A compound condition has one of the following forms:

condition AND condition

condition OR condition

NOT condition

(condition)

The meanings of AND, OR and NOT are given by the following table where A and B represent conditions:

A	B	A AND B	A OR B	NOT A
true	true	true	true	false
false	true	false	true	true
true	false	false	true	false
false	false	false	false	true

Since the compound conditions formed by AND, OR and NOT are conditions, they may be connected by the same logical connectors. Thus, forms such as the following are possible:

A AND B OR C AND D AND NOT E

The meaning of such an expression is given by evaluating all ANDs before ORs. A different interpretation is obtained by enclosing conditions in brackets. The meaning is then obtained by first evaluating all the conditions enclosed in brackets according to the above rules. If brackets are nested, evaluation proceeds from the least inclusive set to the most inclusive set. For example, the expression

A AND ((B OR C) AND D AND NOT E)

is evaluated by first solving (B OR C); then ((B OR C) AND D AND NOT E) and finally the whole expression.

STATEMENT FORMATS

In the formats of the Procedure Division statements which follow in alphabetical order, "literal" is understood to include non-numeric literals, numeric literals and figurative constants unless any are explicitly excluded. Similarly, "identifier" is understood to include data names, qualified data names and subscripted data names, unless any are explicitly excluded. Users should also note that although all the verb formats given may be used with the COBOL compilers, they should check with the table of compiler specifications on pages 5 to 11 that particular options are available to the compiler in use.

ACCEPT

FUNCTION The ACCEPT statement causes low volume data to be read from the console typewriter or from another specified peripheral.

FORMAT 1 ACCEPT identifier [FROM mnemonic-name]

FORMAT 2 ACCEPT FILE KEYS FROM mnemonic-name

NOTES

1 Format 1 without the FROM option indicates that information consisting of one 1900 Series word to be stored in the area corresponding to "identifier" is to be received from the console typewriter.

2 At the point in the execution of the program corresponding to "ACCEPT identifier" the message "COBOL ACCEPT (SEE JOB SHEET) nnnnnn M" is typed on the console typewriter to identify the ACCEPT statement and the program is suspended. nnnnnn is the sequence number of the COBOL sentence and M is the number of the ACCEPT statement within the sentence. The operator types in the value specified in the operating instructions by means of the console directive ALTER, for example:

AL #NAME 0 W

where W is the value. He restarts the program by typing GO #NAME. Note that word 0 is the only word used for this purpose.

3 The value can be typed in one of three ways:

(a) As an octal number. For example, typing *41424344 gives rise to:

100001 100010 100011 100100

representing ABCD. All 8 octal digits must always be typed.

(b) As a positive decimal unsigned integer in the range 0 to 8,388,607 which is converted to its binary equivalent. For example, 12 gives rise to:

000000 000000 000000 001100

(c) A negative decimal integer in the range -1 to -8,388,608 to be input must first be converted to its binary complement equivalent and then input as an octal number. For example, -44 gives rise to:

111111 111111 111111 010100

which is input as *77777724. All 8 octal digits must always be typed.

4 The method used to type in the value does not affect the manner in which it is stored in the area indicated by 'identifier'. If the receiving field is described as ALPHABETIC or ALPHANUMERIC the object program assumes that four characters will be input and moves them into the field, obeying rules for moving ALPHANUMERIC items. On the other hand, if the receiving field is numeric with usage DISPLAY, DISPLAY-n, COMPUTATIONAL or COMPUTATIONAL-1, the program considers the word of information as a SYNCHRONIZED RIGHT COMPUTATIONAL integer and moves it into the field, obeying rules for moving numeric literals.

5 The FROM mnemonic name option allows information to be transferred from the peripheral unit specified or Executive as well as from the console typewriter. The information is stored in the area corresponding to "identifier". See message LOAD ACCEPT DATA ON *nn*, on page 249.

6 The peripheral unit is identified by "mnemonic-name" which is defined by a statement in the SPECIAL-NAMES paragraph of the Environment Division. It must be either a card reader or a paper tape reader.

7 The date and time may be obtained from Executive by specifying

*DATE IS mnemonic-name or

*TIME IS mnemonic-name

in the SPECIAL-NAMES paragraph of the Environment Division.

The date can be obtained as 8 characters in the form dd/mm/yy and the time as 8 characters in the form hh/mm/ss. In these cases "identifier" must be a character group field or an alphanumeric elementary field. If "identifier" is less than or greater than 8 characters, the value is truncated or space-filled on the right.

It is also possible to obtain the date expressed as the number of days since 31st December 1899 (1st January 1900 = day 1). The date will be obtained in this form if "identifier" is a NUMERIC field. The rules for MOVEing a field described as PICTURE 9(6) COMPUTATIONAL will apply when the value is stored in "identifier".

8 If the peripheral unit is a card reader, the ACCEPT can be regarded as a transfer of information from an alphanumeric field which starts at column 1 of the card available for reading. The field is considered to extend along the card, and subsequent cards if necessary, so that it is the same length as the field into which the information is to be accepted unless it is terminated beforehand by a card containing **** in its first four columns. In this case the field is considered to extend only to the last column of the card preceding ****. If the field on the card is thus shorter than the destination, the standard procedure applies for moving information from an alphanumeric field to a larger field. Any other information on the card containing **** or beyond the last column transferred is ignored. The next peripheral transfer from the same card reader starts by reading a new card. The characters accepted from cards must be taken from the character set in Appendix 1

9 If the peripheral unit is a paper tape reader, the ACCEPT can be regarded as a transfer of information from an alphanumeric field which starts at the next character to be read from paper tape. The field extends along the tape so that it is the same length as the field into which the information is to be accepted unless it is terminated beforehand by a newline character. In this case the field is considered to extend only as far as the newline character. If the field on paper tape is thus shorter than the destination, the standard procedure applies for moving information from an alphanumeric field to a longer field. The characters accepted from paper tape must be taken from the character set in Appendix 1 but must not include $] ↑ ←. Data on paper tape must be terminated by a newline character when ACCEPTed into fields which are more than 128 characters in length and which are not multiples of 128 characters.

10 Format 2 is used to read from cards or paper tape information required when opening files. One use is to read in generation numbers for magnetic tape and direct access files, the format for punching which is given on page 143.

ADD

FUNCTION The ADD statement causes two or more numeric operands to be summed and the result stored.

FORMAT 1

```
ADD { identifier-1 } [ ,identifier-2 ]
    { literal-1    } [ ,literal-2    ] ...TO
identifier-n [ROUNDED] [ON SIZE ERROR imperative statement]
```

FORMAT 2

```
ADD { identifier-1 } , { identifier-2 } [ identifier-3 ]
    { literal-1    }   { literal-2    } [ literal-3    ] ...GIVING identifier-n
[ROUNDED] [ON SIZE ERROR imperative statement]
```

FORMAT 3

```
ADD { identifier-1 } [ ,identifier-2 ]
    { literal-1    } [ ,literal-2    ] ...identifier-n
[ROUNDED] [ON SIZE ERROR imperative statement]
```

FORMAT 4

```
ADD { CORRESPONDING } identifier-1 TO identifier-2 [ROUNDED]
    { CORR          }
[ON SIZE ERROR imperative statement]
```

NOTES

1 Each identifier must refer to an unedited elementary numeric field except that identifiers appearing to the right of GIVING may refer to an edited or to an unedited elementary numeric field.

2 Each literal must be a numeric literal.

3 The maximum size of an operand which has usage COMPUTATIONAL-1 is 46 binary digits excluding the sign. The maximum size of any other operand, except of the field named to the right of GIVING, is 12 decimal digits. The field named to the right of GIVING can be the maximum size possible for its class and usage.

4 The ROUNDED option means that the sum is to be rounded if necessary before being stored in the result field. (See Chapter 9.)

5 The SIZE ERROR option means that if the sum, rounded if required, is too large for the result field, the result field is left as it was before the ADD operation and the imperative statement specified in the SIZE ERROR clause is obeyed. If the SIZE ERROR clause is not used and a size error condition occurs, the value of the result field is indeterminate. The exact rules of what constitutes a size error are given in Chapter 9.

6 Detailed rules for arithmetic, including rules for operating on fields with different usages, are given in Chapter 9.

7 If format 1 is used, the values of the operands preceding the word TO are added together; the sum is added to the current value of identifier-n and the result, rounded if necessary and subject to the size error condition, is stored in identifier-n.

8 If format 2 is used, the values of the operands preceding the word GIVING are added together and the sum, rounded if necessary and subject to the size error condition, is stored as the new value of identifier-n.

9 If format 3 is used, the values of the operands are added together and the sum, rounded if necessary and subject to the size error condition, is stored as the new value of identifier-n.

10 If format 4 is used, data items in identifier-1 are added to and stored in corresponding data items in identifier-2 (See also page 67, the CORRESPONDING option).

FUNCTION The ALTER statement modifies a predetermined sequence of operations.

FORMAT ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
[procedure-name-3 TO [PROCEED TO] procedure-name-4]...

NOTES

1 Each procedure-name-1, procedure-name-3 .... is the name of a paragraph which contains only one sentence consisting of a GO TO statement without the DEPENDING option.

2 Each procedure-name-2, procedure-name-4... is the name of a paragraph or section in the Procedure Division.

3 During execution of the object program, the ALTER statement modifies the GO TO statements in the paragraphs named procedure-name-1, procedure-name-3... replacing the objects of the GO TOs by procedure-name-2, procedure-name-4... respectively.

4 A GO TO statement in a section whose priority number is greater than 49 must not be referred to by an ALTER statement in a section with a different priority number. All other uses of the ALTER statement are valid and are performed even if the GO TO to which the ALTER refers is in a segment of the program that has not yet been called for execution (see Chapter 7 on Segmentation).

5 The effect of the ALTER statement will be the same whether or not PROCEED TO is entered after TO; but TO PROCEED or TO TO is invalid.

CALL

FUNCTION The CALL statement transfers control to a subroutine (COBOL or other).

FORMAT CALL subroutine-name [USING identifier-1 [identifier-2...]]

NOTES

1 The CALL statement may be used without the USING clause when referring to subroutines other than COBOL subroutines.

2 Identifier-1... must be the names of records or fields (so long as the latter are word aligned) in the File, Working-Storage, Constant or Linkage Sections.

3 Identifier-1... must be the names of records (01 or 77 levels) in the File, Working-Storage, Constant or Linkage Sections.

4 The number and order of identifiers in the USING clause correspond to the number and order of records in the USING clause of the Procedure Division heading of the called subroutine, if present. Otherwise, they correspond to the number and order of the records of the called subroutine's Linkage Section except in the special case mentioned in note 5 on page 41.

5 The parameters in the USING clause are interpreted as giving addresses of the fields named without any descriptions. The corresponding Linkage Section records in the subroutine give descriptions of the areas starting at these addresses.

6 The subroutine name must not exceed 11 characters.

CLOSE

FUNCTION The CLOSE statement terminates the processing of input and output reels, units and files with optional rewind and/or lock where applicable.

FORMAT

```
CLOSE file-name-1 [REEL] [ WITH { NO REWIND } ]
                         [      { LOCK      } ]
                         [      { RETAIN    } ]

[file-name-2 [REEL] [ WITH { NO REWIND } ] ] ...
                   [      { LOCK      } ]
                   [      { RETAIN    } ]
```

NOTES

1 Each file name is the name of a file upon which the CLOSE statement is to operate.

2 If a file is held on *punched cards or paper tape or the line printer*, the meanings of the various options of CLOSE are as follows:

CLOSE: The unit holding the file is released from the program.
CLOSE WITH LOCK: The unit holding the file is released and the file cannot be referred to again by the program which closed it with lock.

The compiler treats the other options as errors in the source program when applied to punched cards, paper tape or line printer.

3 If a file is held on *magnetic tape*, the meanings of the various options of CLOSE are as follows:

CLOSE: The magnetic tape holding the file or the last reel of the file is rewound and the magnetic tape deck is released from the program.

CLOSE WITH LOCK: This has the same meaning as CLOSE except that the magnetic tape deck holding the file or its last reel becomes unavailable to any program without operator intervention. In addition, the file which has been closed cannot be referred to again by the program which closed it with lock. This applies even if the reel is mounted on a different magnetic tape deck.

CLOSE WITH NO REWIND: This applies only to magnetic tape files. For output files it causes the end label to be written and the tape is not rewound. For input files, end-of-file must have been reached and the AT END clause of a READ statement obeyed before the CLOSE ... NO REWIND is obeyed, otherwise an error message will be printed; the CLOSE statement may even be in the AT END clause. The tape is not rewound.

CLOSE WITH RETAIN: The file is rewound and left on-line to the program. Before the file can then be read or written an appropriate OPEN statement must be obeyed. This option allows the user to process the same physical file several times in a program without using the ordinary CLOSE. There are no console messages. The file is not released from the program and no other program can access the file.

It is not possible to CLOSE ... RETAIN a file being READ in REVERSED mode, after the AT END clause has been obeyed.

CLOSE REEL: The first record of the subsequent reel becomes available when the next READ statement is obeyed if the file was opened for input; otherwise the next WRITE statement causes a continuation reel to be opened and the next record written to it. If dumping at the end of each reel of a file is specified by a RERUN statement, dumping takes place. If CLOSE REEL is applied to the last or only reel of a file a subsequent READ statement causes a message to the operator to be shown.

```
CLOSE REEL WITH { NO REWIND }
               { LOCK      }
               { RETAIN    }
```

In all these cases a message is printed at compile time and the statement is given the same meaning as CLOSE REEL.

4 If a file is held on a direct access device, a CLOSE statement may include the LOCK or RETAIN options, but must not include any of the other options.

A CLOSE statement applied to an output file writes a standard end of file label in the end file bucket if specified and releases the file from the program's reservations.

The effect of LOCK and RETAIN is similar to their effect on magnetic tape files, except that RETAIN means that the file is reset, and left on-line to the program.

5 Input and output files used during a sort must not be open at the time the SORT Statement is obeyed. If they have been OPENed, they must have been CLOSEd, and the CLOSE Statement must not have included the REEL, NO REWIND, LOCK or RETAIN options.

COPY

FUNCTION The COPY statement incorporates library routines into the Procedure Division of the source program.

FORMAT

```
COPY library-name [REPLACING word-1 BY word-2
                  [word-3 BY word-4 ] ... ]
```

NOTE

1 For a full explanation of the COPY statement see Chapter 8 on page 107.

2 Word-n in the format may be any valid COBOL word except a reserved word or any of the words in a NOTE statement.

DELETE

FUNCTION The DELETE statement deletes a logical record from a direct access file and allows the execution of a specified procedure if the contents of the associated ACTUAL KEY are invalid.

FORMAT DELETE record-name [INVALID KEY imperative statement]

NOTES

1 A DELETE statement can be written only for a direct access file which is opened in the input-output mode.

2 A DELETE statement for a file whose access mode is SEQUENTIAL must be preceded by a READ statement for the same file. In this case the effect of the DELETE statement is to delete the last record read. A series of WRITE statements can intervene between the READ and DELETE statements without changing the effect of the DELETE statement.

3 If ACCESS SEQUENTIAL is specified for the file, the INVALID KEY clause need not be specified since the DELETE statement must be preceded by a READ statement which references the same record.

4 If a DELETE statement is applied to a file whose access mode is RANDOM and organization is INDEXED, the following two cases can occur:

   (a) If the DELETE statement is logically preceded by a READ statement without an intervening SEEK statement, the DELETE statement deletes the last record read. The INVALID KEY clause need not be specified for the DELETE statement.

   (b) In other cases the DELETE statement deletes the record located by the preceding SEEK statement.

5 If a DELETE statement is applied to a file whose access mode is RANDOM and organization is DIRECT, the contents of the ACTUAL KEY must first be set to the desired value by using the appropriate address generation procedure. The effect of the DELETE statement is then to delete the record which is specified by the SYMBOLIC KEY field and held in the bucket specified by the ACTUAL KEY field. A READ statement can intervene between the execution of the address generation procedure and the DELETE statement without changing the effect of the DELETE statement.

6 If ACCESS MODE IS RANDOM is specified for a file and a specific record has been referenced, the INVALID KEY procedure is executed when the logical bucket number of the required record lies outside the file limits specified when allocating the file. The logical bucket number of the required record is either indicated by the ACTUAL KEY field or determined by the execution of a SEEK statement.

DISPLAY

FUNCTION The DISPLAY statement causes low volume data to be written on an appropriate hardware device.

FORMAT

```
DISPLAY { identifier-1 } [ { identifier-2 } ] ...
        { literal-1    }   { literal-2    }
        [UPON mnemonic-name]
```

NOTES

1 When the UPON option is not used information is displayed on the console typewriter. When the UPON option is used the output medium is identified by mnemonic-name.

2 The values displayed on the output medium correspond to the identifiers and literals in the DISPLAY statement. Any number of fields and literals may be displayed but the sum of the sizes of the items in any one DISPLAY statement must not exceed 40 characters if displayed on the typewriter.

3 A literal may be any figurative constant except ALL. If a figurative constant is specified, only one occurrence of it is displayed. Numeric literals are displayed as if alphanumeric.

4 A COMPUTATIONAL field is displayed as if it had been first moved to a field with DISPLAY-3 usage and the same Picture, that is, as a decimal number with a preceding sign if required.

5 A COMPUTATIONAL-1 field is displayed in binary form, that is, each bit is displayed as a separate character. If the field is signed, the first bit corresponds to the sign bit.

6 Group fields to be displayed must start on a 1900 Series character boundary and be a whole number of characters long. Each 1900 Series character is displayed as a separate character and is treated as alphanumeric.

7 If the field to be displayed has DISPLAY, DISPLAY-1, DISPLAY-2, DISPLAY-3, DISPLAY-4 or DISPLAY-5 usage each character displayed is the corresponding character in the computer store.

8 Fields are displayed in the same order as in the DISPLAY statement, on the same line and without spacing. Thus if the information to be displayed is to be spaced, spaces must be inserted in the DISPLAY statement as in the following example:

```
DISPLAY "TOTAL-TAX = " TOTAL-TAX.
```

This means the letters TOTAL-TAX are to be typed followed by a space, = , another space and the contents of the field TOTAL-TAX. Thus if TOTAL-TAX has PICTURE 999 and value 012, the information will be typed out in the following format:

```
TOTAL-TAX = 012
```

9 By using the UPON option the user can cause low volume data to be printed on a line printer, punched on cards or punched on paper tape.

10 If the UPON option is used the mnemonic name identifies the type of output device to be used. It must be defined in the SPECIAL-NAMES section of the Environment Division.

11 If information to be displayed on a line printer cannot be contained on one line, it continues on the subsequent lines. If the last or only line is not filled, the remainder is left blank and the next operation involving that printer starts with the next available line. A value is split over two lines if necessary.

12 If information to be punched on a card cannot be contained on one card, it continues on subsequent cards. If the last or only card is not filled, the remainder is left blank and the next operation involving that card punch starts with the next available card. A value is split over two cards if necessary.

13 If information is punched on paper tape, the punch is left positioned to punch the character immediately following the last character of the last value displayed.

14 The DISPLAY statement with the UPON option causes the required peripheral to be allocated to the program if it is available and is not already so allocated. It does not cause the peripheral to be released and unless the same peripheral is used by a file which is subsequently closed, the peripheral will not be released. The DISPLAY statement is implemented in this way for the convenience of the programmer who wishes to use it for intermediate printing while checking a program. If the printer were released after each DISPLAY, the information displayed by successive statements could appear on different printers.

15 If the user wishes to display information upon a peripheral which is assigned to a file, he must cause the mnemonic name to refer to the same kind of peripheral with the same number as that to which the file is assigned. The same peripheral will then be used provided the DISPLAY is carried out before the file is closed. For example, if the Environment Division contains the statements

```
SELECT A ASSIGN TO PRINTER 1
```

and

```
PRINTER 1 IS TEST-PRINTER
```

the statement

```
DISPLAY COUNT UPON TEST-PRINTER
```

obeyed before file A has been closed, whether it has been opened or not, will cause the value of COUNT to be printed on the printer assigned to file A. This situation is to be contrasted with ACCEPT which causes the peripheral to be released after use.

DIVIDE

FUNCTION The DIVIDE statement divides the value of one numeric item into another and sets the value of a numeric field equal to the result.

FORMAT 1

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }
GIVING identifier-3 [ROUNDED] [ON SIZE ERROR imperative statement]
```

FORMAT 2

```
DIVIDE { identifier-1 } INTO identifier-2
       { literal-1    }
[ROUNDED] [ON SIZE ERROR imperative statement]
```

NOTES

1 Each identifier must refer to an unedited elementary numeric field except identifier-3 in format 1 which may refer to an edited or unedited elementary number item.

2 Each literal must be a numeric literal.

3 The maximum size of an operand which has usage COMPUTATIONAL-1 is 46 binary digits excluding the sign. The maximum size of any other operand except identifier-3 is 12 decimal digits plus the sign. Identifier-3 can be the maximum size possible for its class and usage.

4 The ROUNDED option means that the quotient is to be rounded if necessary before being stored in the result field (see Chapter 9).

5 The SIZE ERROR option means that if the quotient, rounded if required, is too large for the result field, the result field is left as it was before the DIVIDE operation and the imperative statement specified in the SIZE ERROR option is obeyed. If the SIZE ERROR option is not used and a size error condition occurs, the value of the result field is undefined. The exact rules of what constitutes a size error are given in Chapter 9.

6 Detailed rules for arithmetic including rules for operating on fields with different usages are given in Chapter 9.

7 If format 1 is used, the initial value of identifier-1 or literal-1 is divided into the initial value of identifier-2 or literal-2 and the quotient, rounded if necessary and subject to the size error condition is stored in identifier-3.

8 If format 2 is used, the initial value of identifier-1 or literal-1 is divided into the initial value of identifier-2 and the quotient, rounded if necessary and subject to the size error condition, is stored in identifier-2.

ENTER

FUNCTION The ENTER statement provides a means of incorporating into COBOL programs library subroutines in semi-compiled form or users' subroutines written in PLAN or FORTRAN and converted into semi-compiled form.

FORMAT ENTER language-name subroutine-name [USING P1 [P2...Pn]]

NOTES

1 The language-name can be any word. It is included to conform with the COBOL standard and because the user might find it useful for documentary purposes. Although it must be present, it is ignored by the compiler.

2 The subroutine-name must not exceed 11 characters.

3 Existing library subroutines in semi-compiled form or users' subroutines written in PLAN or FORTRAN and converted into semi-compiled form may be called in this way. Even if a given subroutine is addressed by several ENTER statements, only one copy of the subroutine will be included in the object program, where it will be placed in the fixed portion (see Chapter 7). The semi-compiled COBOL program and any semi-compiled users' subroutines and/or library subroutines are inserted during the consolidation run, which forms the last part of the compilation. The consolidation run produces as output one program, which is ready to be run via the loader.

4 If the USING option is employed the subroutine must incorporate coding to accept the parameters. The instruction that is generated to call the subroutine uses index register 1. The compiler generates additional instructions following the CALL instruction such that if the subroutine contains the instruction

```
OBEY    n(1)
```

the address of the (n + 1)th parameter is placed in index register 3. Obviously if the subroutine is ENTERed with m parameters, it must end with

```
EXIT    1    m
```

FORTRAN subroutines are produced by the FORTRAN Compiler with the necessary linkage automatically incorporated in them.

5 The parameters P1, P2...Pn may be file-names, record-names, data-names, procedure-names or literals. Data name parameters may be subscripted.

6 FILE-NAME: If the file is a magnetic tape or direct access file, the address loaded on OBEYing the corresponding instruction after the call is that of the File Definition Area corresponding to the file. If the file is stored on any other medium, the address is that of the Peripheral Control Area. In this case, if the file is double-buffered (see RESERVE ALTERNATE AREAS, page 28 the PCA) contains a fifth word which contains the address of the "other buffer". The fourth word contains the address of the "current buffer". Words 4 and 5 are switched during the running of the COBOL object program.

7 RECORD-NAME: The address loaded is that of the first word of the record.

8 OTHER DATA-NAMES: The address loaded is that of the corresponding field. If the field is DISPLAY, the address is in character modifier form. If it is COMPUTATIONAL or COMPUTATIONAL-1 and SYNCHRONIZED, the address is the address of the first word: if not SYNCHRONIZED, the address is a bit address, i.e. the address will be an integer n, meaning the nth bit of the program. Group fields; if it starts on a character boundary and is a whole number of characters long, the address loaded is in character modifier form; otherwise it is a bit address. If a data-name parameter is subscripted and the subscripts are data-names, the address is evaluated each time the corresponding OBEY statement is obeyed (see page 84).

9 PROCEDURE-NAME: The address is that of a pair of words containing the address corresponding to the procedure-name in the standard form required by the overlay package:

(a) if the procedure-name is in an overlay section (i.e. one whose priority is greater than or equal to SEGMENT-LIMIT), the first word of the pair contains "Area number/Unit number". Thus

the first 9 bits contain the Area number (which in COBOL object programs always equals 1) and the last 15 contain the Unit or overlay number.

(b) if the procedure-name is in the fixed section, the first word equals zero.

The second word contains the absolute address corresponding to the procedure-name. This type of parameter can be used to provide different exit points for, say, error conditions.

For obvious reasons of context a paragraph-name parameter must not be numeric nor the same name as a data-name since the first would be treated as a numeric literal and the second as the address of the parameter.

10 LITERALS: The address given is that of the first character of the literal which is stored as follows:

(a) non-numeric literals: the characters between the quotes are stored as a character string.

(b) numeric-literals: the literal, including a sign, if any, is stored as a character string. If the literal contains a period, the period is omitted and the compiler gives a CAUTION since the period has no significance; it is assumed the subroutine is written to accept figures in tenths, hundredths or whatever.

(c) figurative constants: the address given is that of a stored zero or space character for ZEROS or SPACE respectively.

FURTHER NOTES

The above is all the user really needs to know about how the different parameters are treated. However, for the user who is interested, the following additional points concerning file-name, record-name and subscripted parameters may be of some interest

(i) SUBSCRIPTED DATA-NAME PARAMETERS As stated on page 83, if the nth parameter of a subroutine is a data-name, then "OBEY n(1)" will give the address of the data-name field in X3. If the data-name is subscripted and the subscripts are data-names, "OBEY n(1)" will cause the address of the *current* table item indicated by the *current* values of the data-name subscripts to be placed in X3. In this case, the OBEY causes a subroutine in the calling sequence to be executed which calculates the current address. The user should note that this subroutine may destroy X0. The normal procedure in ENTERed subroutines is to sort out all the addresses at the start of the subroutine, dump them, and then use the dumped addresses as required. However, the above system does give the user a facility for manipulating tables easily. For example, suppose the subroutine has to process a one-dimensional table called TAB.

```
01 TAB
02 ELEMENT PICTURE 999 OCCURS 10 TIMES
```

The user would write:

```
ENTER PLAN SUB USING TAB.
```

The subroutine would then be written to pick up the address of TAB, the start of the table, and to check through it three characters at a time, i.e. the subroutine has to work out how to calculate subscripted field addresses. However, the user could define another field, say:

```
77 P PICTURE 99 COMP SYNC RIGHT.
```

and then write:

```
ENTER PLAN SUB USING ELEMENT (P), P.
```

The subroutine would then read:

```
Obtain address of P. (OBEY 1(1))
Set P = 1
┌─→ Obtain address of ELEMENT (P). (OBEY 0(1))
│   Operate on it.
│   Using address of P, Add 1 to it.
No  Have we finished?
└─
    Exit
```

Thus, by leaving the OBEY corresponding to the table element till later, the fact that the addresses of subscripted items are calculated can be used to reflect the current value of the subscripts. This example may seem trival, but subscripting is not as simple as it seems, particularly when the table is structured or there are 3 levels of subscripting. Moreover, the way the compiler generates code to calculate subscript addresses is optimized depending on considerations such as whether all the elements of the table start on the same bit within a word. Using the above technique, the user is effectively provided with an optimized subroutine to calculate table addresses.

(ii) FILE-NAME PARAMETERS It was explained above that an OBEY addressing a file-name parameter gives the address of the corresponding File Definition or Peripheral Control Area. However, considerably more information can be obtained about the file, the peripheral to which it is assigned and its buffers. If instead of "OBEY n(1)" the user writes

```
LDX 2 n(1)
```

then writing

```
LDX  3  0(2) gives X3 = address of FDA or PCA
LDX  3  1(2) gives X3 = "Unit number/Buffer address"
LDX  3  2(2) gives X3 = "Number of buffers/Size of buffers"
LDX  3  3(2) gives X3 = address of record area.
```

Thus the first instruction gives the same as "OBEY n(1)". The second gives the unit number in bits 0-8 and the buffer address in the last 15 bits. The number of buffers is 1 or 2. The size of buffers is in words for fast peripherals and in characters for slow peripherals.

(iii) RECORD-NAME PARAMETERS Similar additional information is obtainable from record parameters. If the record is in the FILE SECTION, assuming the user writes

```
LDX  2  n(1)
LDX  3  0(2) gives X3 = address of record area
LDX  3  1(2) gives X3 = length of record in words
LDX  3  2(2) gives X3 = address of FDA or PCA
LDX  3  3(2) gives X3 = "Unit number/Buffer address"
LDX  3  4(2) gives X3 = "Number of buffers/Size of buffers"
```

If the record is in the WORKING-STORAGE SECTION

```
LDX  3  0(2) gives X3 = address of record area
```

Other LDX's give unpredictable results so there is no point in using other than "OBEY n(1)" in this case.

The type number of the peripheral on which a file is held is contained in Bits 1 to 8 of the first word of the File Definition or Peripheral Control Area. To extract the type number, given a record-name parameter right-justified in, for example, X4, the user must write

```
LDX  2  n(1)
LDX  3  2(2)
LDXC 4  0(3)
SRL  4  15
```

If the parameter is a file-name parameter, the sequence is the same except that the second instruction is

```
LDX  3  0(2)
```

EXAMINE

FUNCTION The EXAMINE statement replaces or counts the number of occurences of a given character in a data item.

FORMAT

```
                                  ┌ TALLYING { UNTIL FIRST } literal-1 [REPLACING
                                  │          { ALL         }
                                  │          { LEADING     }   BY literal-2]
EXAMINE identifier                │
                                  │ REPLACING { ALL          }
                                  │           { LEADING      } literal-3 BY literal-4
                                  └           { [UNTIL] FIRST }
```

NOTES

1 Examination of identifier proceeds as follows:

(a) If the data item referred to by the EXAMINE statement is non-numeric, examination starts at the left-most character and proceeds to the right. Each character in the data item specified by the identifier is examined in turn.

(b) If the data item referred to by the EXAMINE statement is numeric it must consist of numeric characters and may have an operational sign. Examination starts at the left most character (excluding the sign) and proceeds to the right. Each character except the sign is examined in turn. Regardless of where the sign is physically located, it is completely ignored by the EXAMINE statement.

2 The TALLYING option creates an integral count which replaces the value of a special register called TALLY. The count represents the number of:

(a) Occurrences of literal-1 when the ALL option is used.

(b) Occurences of literal-1 prior to encountering a character other than literal-1 when the LEADING option is used.

(c) Characters not equal to literal-1 encountered before the first occurence of literal-1 when the UNTIL FIRST option is used.

3 When either of the REPLACING options is used the replacement rules are as follows, subject to Note 2.

(a) When the ALL option is used, literal-2 or literal-4 is substituted for each occurrence of literal-1 or literal-3.

(b) When the LEADING option is used, substitution of literal-1 or literal-3 by literal-2 or literal-4 terminates as soon as a character other than literal-1 or literal-3 or the right-hand boundary of the data item is encountered.

(c) When the UNTIL FIRST option is used, the substitution of characters not equal to literal-1 or literal-3 by literal-2 or literal-4 terminates as soon as literal-1 or literal-3 or the right-hand boundary of the data item is encountered.

(d) When the FIRST option is used the first occurrence of literal-1 or literal-3 is replaced by literal-2 or literal-4.

4 Identifier must have USAGE DISPLAY.

5 Each literal must consist of a single character belonging to a class consistent with that of identifier; in addition, each literal may be any figurative constant, except ALL.

FUNCTION The EXIT statement provides a common end point for a series of procedures.

FORMAT 1 EXIT

FORMAT 2 EXIT PROGRAM

NOTES

1 The EXIT statement must appear in a sentence by itself.

2 A format 1 EXIT sentence must be preceded by a paragraph name and must be the only sentence in the paragraph.

3 It is sometimes necessary to transfer control to the end point of a series of procedures. This is normally done by transferring control to the next paragraph or section, but in some cases this does not have the required effect. For instance, the point to which control is to be transferred may be at the end of a range of procedures governed by a PERFORM. The format 1 EXIT statement is provided to enable a procedure-name to be associated with such a point.

4 If control reaches a format 1 EXIT paragraph and no associated PERFORM statement is active, control passes through the EXIT point to the first sentence of the next paragraph.

5 The EXIT PROGRAM statement of format 2 effects return of control from a COBOL subroutine to the point in the calling segment after the call of the subroutine.

GO TO

FUNCTION The GO TO statement causes control to be transferred from one part of the Procedure Division to another.

FORMAT 1 GO TO [procedure-name-1]

FORMAT 2 GO TO procedure-name-1 [procedure-name-2]
... procedure-name-n DEPENDING ON identifier

NOTES

1 Each procedure name is the name of a paragraph or section in the Procedure Division of the program.

2 Whenever a GO TO statement with format 1 is executed, control is transferred to procedure-name-1 or to another procedure name if the GO TO statement has been altered by an ALTER statement.

3 If procedure-name-1 is not specified, an ALTER statement referring to this GO TO statement must be executed before the GO TO statement is executed.

4 If a GO TO statement with format 1 appears in an imperative statement, it must appear as the only or last statement in a sequence of imperative statements.

5 A GO TO statement represented by format 2 causes control to be transferred to one of the procedures named procedure-name-1, procedure-name-2 etc. depending on the value of the identifier being 1, 2, ... n respectively. If the value of the identifier is anything other than the positive integers 1,2, ...n, control is passed to the next sentence.

6 Identifier is the name of a numeric elementary field defined in the Data Division as having no positions to the right of the assumed decimal or binary point.

FUNCTION

The IF statement causes a condition (see pages 68 to 72) to be evaluated. The subsequent action of the object program depends on whether the value of the condition is true or false.

FORMAT

```
IF condition THEN { statement-1      } { OTHERWISE }
                  { NEXT SENTENCE    } { ELSE      }

                  { statement-2      }
                  { NEXT SENTENCE    }
```

NOTES

1 If the condition is true, either statement-1 or the NEXT SENTENCE immediately following the condition is obeyed. If the condition is false, statement-2 or the NEXT SENTENCE immediately following ELSE or OTHERWISE is obeyed.

2 The words ELSE and OTHERWISE are synonymous.

3 The words ELSE NEXT SENTENCE can be omitted if they are the last words in the sentence.

4 If the words NEXT SENTENCE are written, control passes explicitly to the next sentence. This is the next sentence written unless the current sentence is the last sentence of a procedure being performed. In this case the meaning of "next sentence" depends upon the PERFORM verb (see page 94).

5 Statement-1 and statement-2 may contain an IF statement. In this case the word ELSE is considered to belong to the nearest preceding IF which does not have an ELSE. For example, in the statement

IF A = B IF X = Y ADD P TO Q ELSE ADD R TO S ELSE ADD T TO U.

P is to be added to Q if A = B and X = Y,

R is to be added to S if A = B but X is not equal to Y,

T is to be added to U if A is not equal to B.

6 The end of statement-1 is indicated by either a full stop or one of the words ELSE or OTHERWISE which does not belong to a statement forming part of statement-1.

7 The end of statement-2 is indicated by either a full stop or one of the words ELSE or OTHERWISE which does not belong to a statement forming part of statement-2.

MOVE

FUNCTION The MOVE statement transfers data to one or more data areas in accordance with the rules of editing.

FORMAT 1

```
MOVE { identifier-1 } TO identifier-2 [identifier-3] ... [identifier-n]
     { literal      }
```

FORMAT 2

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

NOTES

1 Identifier-1 or literal represents the sending area and identifier-2 . . . the receiving area.

2 The identifiers must not be file names.

3 If format 2 is used, selected items within identifier-1 are moved to selected items within identifier-2 (see also page 67, the CORRESPONDING option). Only one identifier may appear to the right of the word TO and the results are the same as if the programmer had referred to each pair of corresponding identifiers in separate MOVE statements.

4 The rules for moving data are described in detail in Chapter 10.

FUNCTION The MULTIPLY statement causes numeric data items to be multiplied and the result stored.

FORMAT 1

```
MULTIPLY { identifier-1 } BY identifier-2 [ROUNDED]
         { literal-1    }
[ON SIZE ERROR imperative-statement;]
```

FORMAT 2

```
MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }
GIVING identifier-3 [ROUNDED] [ON SIZE ERROR imperative-statement]
```

NOTES

1 Each identifier must refer to an unedited elementary numeric field with the exception that identifier-3 can refer to an edited or unedited elementary numeric field.

2 Each literal must be a numeric literal.

3 The maximum size of an operand which has usage COMPUTATIONAL-1 is 46 binary digits excluding the sign. The maximum size of any other operand except identifier-3 is 12 decimal digits plus the sign. Identifier-3 is not restricted in this way and can be the maximum size possible for its class and usage.

4 The ROUNDED option means that the product is to be rounded if necessary before being stored in the result field. (See Chapter 9 for the exact rules of rounding.)

5 If the SIZE ERROR option is used and the product, rounded if specified, is too large for the result field, the result field is left as it was before the MULTIPLY operation and the imperative statement specified in the SIZE ERROR option is obeyed. If the SIZE ERROR option is not used and a size error condition occurs, the value of the result field is undefined. (See Chapter 9 for the rules of what constitutes a size error condition.)

6 If format 1 is used, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2 and the result, rounded if necessary and subject to the size error condition, is stored in identifier-2. If format 2 is used, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2 or literal-2. The result, rounded if necessary and subject to the size error condition, is stored in identifier-3.

7 Detailed rules of arithmetic, including those dealing with the effect of performing arithmetic on fields with different usages, are given in Chapter 9.

NOTE

FUNCTION The NOTE sentence allows the programmer to write commentary which will be produced on the listing but which will not be compiled.

FORMAT NOTE character-string.

NOTES

1 Any combination of the characters in the character set given in Appendix 1 may be included in the character-string.

2 If NOTE is the first word of a paragraph after the header, the whole paragraph is taken to be commentary and is not compiled.

3 • If a NOTE sentence appears as other than the first sentence of a paragraph, the commentary ends with the first instance of a full stop followed by a space.

FUNCTION The OPEN statement initiates the processing of input, output and input-output files. It also performs checking and/or writing of labels.

FORMAT

```
OPEN { INPUT file-name-1 [REVERSED] [file-name-2 [REVERSED] ...]
       [ OUTPUT file-name-3 [file-name-4] ... ]
     | OUTPUT file-name-1 [file-name-2]
     | [ INPUT file-name-3 [REVERSED] [file-name-4 [REVERSED] ]...]
     | INPUT-OUTPUT file-name-1 [file-name-2]...
     | I-O file-name-1 [file-name-2]...
     | OUTPUT file-name-1 [file-name-2 ]... AT END }
```

NOTES

1 At least one of the options INPUT, OUTPUT, INPUT-OUTPUT or I-O must be specified. However, the same file name cannot appear under more than one of the options. These options may appear in any order.

2 The OPEN statement must be applied to all files and must be executed prior to the first DELETE, READ, REWRITE, SEEK or WRITE statement for that file. A second OPEN statement for a file cannot be executed before the execution of a CLOSE statement for that file.

3 When a file which is held on *punched cards or paper tape or the line printer* is opened, a message is typed requesting the operator to load the file on the appropriate peripheral. While the file is open, the peripheral assigned to the file is the program's number n unit of the required kind, where n is the number quoted after the hardware name in the SELECT... ASSIGN statement (see page 27). If the file is closed and later reopened, a different unit may become unit number n, depending upon the action of Executive and the operator. Executive selects a unit and the operator may change it if he wishes.

4 When a file which is held on *magnetic tape* is opened for *input*, the program searches the header labels of the tapes. The identification fields of the header labels are checked to find the required file as are also the generation number fields if the programmer has specified that this is to be done. If the file required is not found, the program is suspended and a message typed for the operator who may mount the tape and cause the program to continue. The magnetic tape deck prepared by the operator or found by the program becomes the unit allocated to the file until it is closed. When a magnetic tape file is opened for *output*, the program searches for a tape whose header label indicates that it may be written on. If no deck is available, a message to this effect is typed for the operator. Provided a deck is assigned to the file, a standard header label is written on the tape. The label contains the identification and retention period specified in the VALUE OF clause of the file (see page 38) and the file's generation number if required (see page 37).

5 The INPUT-OUTPUT option permits the opening of a direct access file for both input and output operations. It can be used only for files held on direct access devices.

6 When a *direct access* file is opened the opening procedures performed are the same as those for magnetic-tape files. If the file is opened for *input* or *input-output*, as with a magnetic tape file, a check is made on the generation number of the file if LABEL RECORDS ARE STANDARD WITH GENERATION-NO has been specified. An ACCEPT FILE KEYS statement can be used to insert the generation number but if one has not been obeyed when the file is opened the object program requests the generation number from the operator. If a check on the generation number has not been specified, the file in the system with the specified label and the highest generation number is opened. If a file is opened for *output* two generation numbers may be specified by specifying LABEL RECORDS ARE STANDARD WITH GENERATION-NO and by obeying an ACCEPT FILE KEYS statement. If an ACCEPT FILE KEYS statement has not been obeyed before the OPEN statement, the file in the system with the specified label and the highest generation number is opened and the new generation number to be written to the file is requested from the operator by the object program.

7 The AT END option can only be used for direct access files stored serially. It permits the writing of new information at the end of the file.

8 OPEN INPUT....REVERSED can be applied only to magnetic tape files that are in a "CLOSE...NO REWIND" state. READ statements applied to the file will cause records to be presented in the reverse order to that in which they were written to the file, and the AT END clause will be obeyed when the *beginning* of the file is reached. The file can then be closed with CLOSE or CLOSE..... LOCK

The clause cannot be used with multi-reel files. If it is, only the reel to which CLOSE...NO REWIND has been applied will be read.

9 It is the user's responsibility to ensure that the magnetic tape transport holding a file to which OPEN... REVERSED is to be applied, does in fact have the Read Reverse facility available.

PERFORM

FUNCTION — The PERFORM statement is used to depart from the normal sequence of execution in order to execute one or more procedures either a specified number of times or until a specified condition has been satisfied and to return control to the normal sequence.

FORMAT 1

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

FORMAT 2

```
PERFORM procedure-name-1 [THRU procedure-name-2] { identifier-1 } TIMES
                                                 { integer-1    }
```

FORMAT 3

```
PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1
```

FORMAT 4

```
PERFORM procedure-name-1 [THRU procedure-name-2]
    VARYING identifier-1 FROM { literal-2    } BY { literal-3    } UNTIL condition-1
                              { identifier-2 }    { identifier-3 }
    [AFTER identifier-4 FROM { literal-5    } BY { literal-6    } UNTIL condition-2
                             { identifier-5 }    { identifier-6 }
    [AFTER identifier-7 FROM { literal-8    } BY { literal-9    } UNTIL condition-3] ]
                             { identifier-8 }    { identifier-9 }
```

NOTES

1 Each procedure-name is the name of a section or paragraph in the Procedure Division.

2 Each identifier represents an unedited numeric elementary field described in the Data Division. In format 2 the identifier represents an elementary unedited numeric field defined as having no positions to the right of the assumed decimal or binary point.

3 Each literal represents an integral numeric literal.

4 When the PERFORM statement is executed, control is transferred to the first statement of the procedure named procedure-name-1. An automatic return to the statement following the PERFORM statement is established as follows:

   (a) If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, the return is after the last sentence of procedure-name-1.

   (b) If procedure-name-1 is a section name and procedure-name-2 is not specified the return is after the last sentence of the last paragraph in procedure-name-1.

   (c) If procedure-name-2 is specified and is a paragraph-name, the return is after the last sentence of the paragraph.

   (d) If procedure-name-2 is specified and is a section-name, the return is after the last sentence of the last paragraph in the section.

5 There is no necessary relationship between procedure-name-1 and procedure-name-2 except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. In particular, GO TO and PERFORM statements can occur between procedure-name-1 and the end of procedure-name-2. If there are two or more direct paths to the return point, then procedure-name-2 must be the name of a paragraph consisting of the EXIT statement to which all of these paths must lead.

6 If control passes to these procedures by means other than a PERFORM statement, it passes through the last statement of the procedure to the following statement as if no PERFORM statement mentioned the procedures.

7 If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedure associated with the included PERFORM must itself be totally

included in or be totally excluded from the logical sequence referred to by the first PERFORM. In other words the sequence of procedures referred to by the PERFORMS must neither overlap nor have a common exit point.

8 Format 1 is the basic PERFORM statement. A procedure referred to by this type of PERFORM statement is executed once and then control passes to the statement following the PERFORM statement.

9 When format 2 is used, the procedures are performed the number of times specified by identifier-1 or integer-1. This must be a positive integer or zero. Control is then transferred to the statement following the PERFORM statement. If the value is zero, control passes immediately to the statement following the PERFORM statement. Once the PERFORM statement has been initiated, any reference to identifier-1 has no effect in varying the number of times the procedures are executed.

10 When format 3 is used, the procedures are performed until the condition specified by the UNTIL option is true. Control is then transferred to the statement following the PERFORM statement. If the condition is true when control passes to the PERFORM statement, the specified procedure is not executed.

The COBOL format

PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1

is equivalent to

P. IF NOT (condition-1) PERFORM procedure-name-1 [THRU procedure-name-2]
GO TO P.

11 Format 4 is used to augment or reduce the value of one or more identifiers in an orderly fashion during the execution of a PERFORM statement.

The format

PERFORM procedure-name-1 [THRU procedure-name-2] VARYING identifier-1
FROM { literal-2 / identifier-2 } BY { literal-3 / identifier-3 } UNTIL condition-1

is equivalent to

MOVE { literal-2 / identifier-2 } TO identifier-1.

P1. IF NOT (condition-1) PERFORM procedure-name-1 [THRU procedure-name-2]

ADD { literal-3 / identifier-3 } TO identifier-1 GO TO P1.

The format

PERFORM procedure-name-1 [THRU procedure-name-2]
VARYING identifier-1 FROM { literal-2 / identifier-2 } BY { literal-3 / identifier-3 } UNTIL condition-1
AFTER identifier-4 FROM { literal-5 / identifier-5 } BY { literal-6 / identifier-6 } UNTIL condition-2

is equivalent to

MOVE { literal-2 / identifier-2 } TO identifier-1.

P1. IF NOT (condition-1) PERFORM P2 THRU P3

ADD { literal-3 / identifier-3 } TO identifier-1 GO TO P1.

where P2 and P3 are consecutive paragraphs as follows:

P2. MOVE { literal-5 / identifier-5 } TO identifier-4.

P3. IF NOT (condition-2) PERFORM procedure-name-1 [THRU procedure-name-2]
ADD { literal-6 / identifier-6 } TO identifier-4 GO TO P3.

The format

PERFORM procedure-name-1 [THRU procedure-name-3]
VARYING identifier-1 FROM { literal-2 / identifier-2 } BY { literal-3 / identifier-3 } UNTIL condition-1
AFTER identifier-4 FROM { literal-5 / identifier-5 } BY { literal-6 / identifier-6 } UNTIL condition-2
AFTER identifier-7 FROM { literal-8 / identifier-8 } BY { literal-9 / identifier-9 } UNTIL condition-3

is equivalent to

MOVE { literal-2 / identifier-2 } TO identifier-1.

P1. IF NOT (condition-1) PERFORM P2 THRU P3
ADD { literal-3 / identifier-3 } TO identifier-1 GO TO P1.

where P2 and P3 are consecutive paragraphs as follows:

P2. MOVE { literal-5 / identifier-5 } TO identifier-4.

P3. IF NOT (condition-2) PERFORM P4 THRU P5
ADD { literal-6 / identifier-6 } TO identifier-4 GO TO P3.

and where P4 and P5 are consecutive paragraphs as follows:

P4. MOVE { literal-8 / identifier-8 } TO identifier-7.

P5. IF NOT (condition-3) PERFORM procedure-name-1 [THRU procedure-name-2]
ADD { literal-9 / identifier-9 } TO identifier-7 GO TO P5.

12 A PERFORM statement can have within its range any number of sections of any priority.

13 When a procedure-name in a segment with a priority number greater than 49 is referred to by a PERFORM statement contained in a segment with a different priority number, the segment referred to is made available in its initial state for each execution of the PERFORM statement.

FUNCTION

1 For sequential files, the READ statement makes available the next logical record from an input file and allows performance of a specified imperative statement when the end of the file is detected.

2 For random file processing, the READ statement makes available a specific record from a direct access file and allows performance of a specified imperative statement if the contents of the associated ACTUAL KEY field are found to be invalid.

FORMAT 1

READ file-name RECORD [INTO identifier] AT END imperative-statement

FORMAT 2

READ file-name RECORD [INTO identifier] INVALID KEY imperative-statement

NOTES

1 Format 1 is used only for non-direct access files and for direct access files when accessed sequentially. Format 2 is used only when accessing specific records in the random access mode.

2 An OPEN statement must be executed for a file prior to the execution of the first READ statement for that file.

3 If, during the execution of a READ statement in format 1, the logical end of the file is reached and an attempt is made to read that file, the statement specified in the AT END phrase is executed. After the execution of the imperative statement of the AT END phrase, a READ statement for that file must not be given without prior execution of a CLOSE statement and an OPEN statement for that file. A file can be opened and closed any number of times in the same program.

4 The INTO option can be used only if the file contains records of one type only, that is, an 01 level entry appears only once in the file definition of the file. If the option is specified, the effect is as if the statement

MOVE record-name TO identifier

immediately followed a READ statement without the INTO option. "READ A INTO B" has the same meaning as "READ A; MOVE A TO B". The record read is then available in both the data area associated with identifier and the input record area.

5 If the end of a tape reel or a direct access unit is recognized during the execution of a READ statement, the program carries out the necessary actions to ensure that the next reel or unit is made available. The programmer need not be aware when writing the Procedure Division that the file occupies more than one reel or unit.

6 User sentinels on magnetic tape are ignored when reading magnetic tape files. Memory dumps associated with the RERUN facility (see page 32) are ignored in a magnetic tape file which is being read.

7 If a direct access file's access mode is SEQUENTIAL and the file is stored serially, a READ statement makes available the record which is in the next physical position on the file. If, however, the file is stored sequentially a READ statement makes available the record which is next in key-sequence order.

8 Format 2 of the READ statement must be used:

(a) for direct access files whose access mode is RANDOM and organization is INDEXED, when it is required to access specific records;

(b) for direct access files whose access mode is RANDOM and organization is DIRECT.

9 If a direct access file's access mode is RANDOM and organization is INDEXED, a READ statement may be preceded by a SEEK statement. Its records can be accessed either randomly or sequentially starting at a specified record.

To access a specified record the key of the desired record must first be set in the SYMBOLIC KEY field. The SEEK statement determines the logical bucket number of the record from the index tables and the READ statement searches this bucket for the record specified by the SYMBOLIC KEY field.

To access a sequential group of records starting at a specified record the specified record is first accessed by the method described above. Provided there are no intervening SEEK statements, subsequent READ statements access the records next in key-sequence order.

10 If a direct access file's access mode is RANDOM and organization is DIRECT, a READ statement makes available a specified record of the file. The key of the desired record must first be placed in the SYMBOLIC KEY field and the appropriate address generation procedure applied to determine the logical bucket number of the record and set it in the ACTUAL KEY field. The READ statement searches the bucket specified in the ACTUAL KEY field for the record with the key specified in the SYMBOLIC KEY field.

11 The INVALID KEY procedure is executed when the logical bucket number of the required record lies outside the file limits specified when allocating the file. The logical bucket number of the required record is either indicated by the ACTUAL KEY field or determined by the execution of a SEEK statement.

12 If the physical block held on an external medium associated with a file holds more than one record, the programmer controls its size by the BLOCK CONTAINS clause (see page 37). READ statements cause new blocks to be obtained as required without any indication in the Procedure Division that this is to be done.

RELEASE

FUNCTION The RELEASE statement transfers records to the initial phase of a sort operation.

FORMAT RELEASE record-name [FROM identifier]

NOTES

1 A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement.

2 Record-name must be the name of a logical record of the associated sort-file and may be qualified.

3 The RELEASE statement causes the record named by record-name to be released to the initial phase of a sort operation.

4 If the FROM option is used, the effect is as if the statement – MOVE identifier TO record-name – immediately preceded a RELEASE statement without the FROM option.

5 After the RELEASE is executed; the record is no longer available. When control passes from the input procedure, the sort file consists of all those records that were placed in it by the execution of RELEASE statements.

RETURN

FUNCTION The RETURN statement obtains sorted records from the final phase of a sort operation.

FORMAT RETURN file-name RECORD [INTO identifier] AT END imperative-statement.

NOTES

1 File-name must be described by a Sort File Description entry in the Data Division.

2 A RETURN statement may only be used within the range of an output procedure associated with a SORT statement.

3 The execution of the RETURN statement causes the next record, in the order specified by the keys listed in the SORT statement, to be made available for processing in the record area associated with file-name.

4 The INTO option can only be used if the sort-file contains records of one type only. If the option is specified, the effect is as if the statement

MOVE record-name to identifier

immediately followed a RETURN statement without the INTO option.

5 After execution of the imperative-statement in the AT END phrase, no RETURN statements may be executed within the current output procedure.

REWRITE

FUNCTION The REWRITE statement replaces a logical record on a direct access device by a specified record.

FORMAT REWRITE record-name [FROM data-name]

NOTES

1 A REWRITE statement can be written only for a file opened in the input-output mode. A READ statement must be executed for the file before a REWRITE statement for the file can be executed.

2 The effect of a REWRITE statement is to replace the last record read by the record specified.

3 If ACCESS MODE IS SEQUENTIAL is specified a series of WRITE statements can intervene between the READ and REWRITE statements without changing the effect of the REWRITE statement. Similarly, if ACCESS MODE IS RANDOM and ORGANIZATION IS INDEXED are specified, a series of WRITE statements can intervene between the READ and REWRITE statements without changing the effect of the REWRITE statement. Otherwise the REWRITE statement must be immediately preceded by the READ statement.

4 If a REWRITE statement is applied to a file which is stored serially and the replacement causes an expansion of the record, the replacement will be carried out only if the bucket is large enough to hold the expanded record. If it is not, an area overflow exception condition will arise.

5 If the FROM option is used, data-name must not be the name of an item in a file containing record-name. This form of the REWRITE statement is equivalent to the following statements:

MOVE data-name TO record-name.

REWRITE record-name.

SEEK

FUNCTION The SEEK statement initiates the accessing of a direct access data record for subsequent reading or writing.

FORMAT SEEK file-name RECORD

NOTES

1 A SEEK statement must be applied only to direct access files whose access mode is RANDOM. If the organization of such a file is INDEXED, a SEEK statement must be written before every DELETE, READ and WRITE statement when accessing specific records. Its effect is to cause the direct access system to transform the contents of the SYMBOLIC KEY field into the logical bucket number of the required record.

2 The execution of a SEEK statement involves the reading of the index tables into store. Normally this is done by reading the tables bucket by bucket into the buffer(s) allocated to the file every time a SEEK statement is obeyed for a particular file. If the number of specific records accessed during the processing of the file is large, the inspection of the index tables by this method can be time consuming. In this case one or two areas which will hold the index tables can be associated with the file by means of an APPLY statement in the I-O CONTROL paragraph. If each of the areas is large enough to hold the corresponding index table, the index tables are read into store on the first access of a specific record and are assumed to be in these areas on subsequent SEEK statements for the file being obeyed. Even if each of the areas is not large enough to hold the corresponding index table, the time spent in inspecting the index tables is proportionally reduced.

3 If the organization of the file is DIRECT, a SEEK statement has no effect and can be included or omitted at will.

4 Two SEEK statements for the same direct access file may logically follow each other. Any validity check associated with the first SEEK statement is negated by the execution of a second SEEK statement.

FUNCTION The SORT statement creates a sort-file by executing input procedures or by transferring records from another file, sorts the records in the sort-file on a set of specified keys, and in the final phase of the sort operation makes available each record from the sort-file, in sorted order, to some output procedure or to an output file.

FORMAT

```
SORT file-name-1 ON { DESCENDING } KEY data-name-1 [,data-name-2] ....
                    { ASCENDING  }

[ ON { DESCENDING } KEY data-name-3 [,data-name-4] .... ] ....
     { ASCENDING  }

{ INPUT  PROCEDURE IS section-name-1 [THRU section-name-2] }
{ USING  file-name-2                                       }

{ OUTPUT PROCEDURE IS section-name-3 [THRU section-name-4] }
{ GIVING file-name-3                                       }
```

SYNTAX RULES

1 File-name-1 must be described in a Sort File Description entry in the Data Division. Each data-name must represent data items described in records associated with file-name-1.

2 File-name-2 and file-name-3 must be described in a File Description entry, not in a Sort File Description entry, in the Data Division.

3 The data-names following the word KEY are listed from left to right in the SORT statement in order of decreasing significance without regard to how they are divided into KEY clauses. The data-names may be qualified but not subscripted.

INPUT AND OUTPUT PROCEDURES

1 General rules

(a) Procedures must not contain SORT statements.

(b) Procedures must not contain any explicit transfers of control to points outside themselves i.e. ALTER, GO TO and PERFORM statements within a procedure are not permitted to refer to procedure names outside the procedure.

(c) Control must not be passed to a procedure except when a related SORT statement is being executed.

(d) Procedures must consist of one or more sections that appear contiguously in the source program and do not form part of any opposite procedure.

2 Input Procedures

(a) If an input procedure is specified, control is passed to it before the sort-file (file-name-1) is sequenced by the SORT statement. When control passes from the last statement in the input procedure, the records that have been RELEASEd to file-name-1 are sorted.

(b) The input procedure must contain at least one RELEASE statement in order to transfer records to the sort-file.

3 Output Procedures

(a) If an output procedure is specified, control passes to it after the sort-file (file-name-1) has been sequenced by the SORT statement. When control passes from the last statement in the output procedure, the sort is terminated and control is passed to the next statement after the SORT statement.

(b) The output procedure must include at least one RETURN statement in order to make sorted records available for processing.

INPUT AND OUTPUT FILES

1 If the USING option is specified, all the records in file-name-2 are considered to be the sort-file. At the time of execution of the SORT statement, file-name-2 must not be open. At the termination of the SORT statement, file-name-2 will be closed.

2 If the GIVING option is used, all the sorted records are automatically transferred to file-name-3 as the implied output procedure for the SORT statement. At the time of execution of the SORT statement, file-name-3 must not be open. At the termination of the SORT statement, file-name-3 will be closed.

NOTES

1 Segmentation can be applied to the sections within the input or output procedures.

2 The Procedure Division may contain more than one SORT statement except that there may not be both a magnetic tape and a direct access sort.

The notes given here are only format rules; for how SORTing has been implemented see page 254.1.

STOP

FUNCTION The STOP statement halts the object program either permanently or temporarily.

FORMAT

```
STOP { literal }
     { RUN   }
```

NOTES

1 The STOP literal format causes the literal to be displayed upon the console typewriter and the program to be suspended. The program can be restarted at the next statement.

2 The literal may be numeric or non-numeric or any figurative constant except ALL. It must not be more than 40 characters in length.

3 The STOP RUN format causes a check to be made that all files are closed and the program to be suspended. The program can be restarted only by transferring control to its entry point.

4 If the STOP RUN format appears in an imperative statement, it must be the only or last statement in a sequence of imperative statements.

SUBTRACT

FUNCTION

The SUBTRACT statement is used to subtract a numeric data item or the sum of two or more numeric data items from a numeric item and to set the value of a field equal to the result.

FORMAT 1

```
SUBTRACT { literal-1    } [ literal-2    ] .... FROM identifier-m [ROUNDED]
         { identifier-1 }   [ identifier-2 ]
[ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
SUBTRACT { literal-1    } { literal-2    } .... FROM [ literal-m    ] GIVING
         { identifier-1 }   { identifier-2 }        [ identifier-m ]

identifier-n [ROUNDED] [ON SIZE ERROR imperative-statement]
```

FORMAT 3

```
SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2 [ROUNDED]
         { CORR          }

[ON SIZE ERROR imperative statement]
```

NOTES

1 Each identifier except identifier-n must refer to an unedited elementary numeric field. Identifier-n, which appears to the right of the word GIVING, may refer to an edited or unedited elementary numeric field.

2 Each literal must be a numeric literal.

3 The maximum size of an operand which has usage COMPUTATIONAL-1 is 46 binary digits excluding the sign. The maximum size of any other operand except identifier-n is 12 decimal digits plus the sign. Identifier-n is not restricted in this way and can be the maximum size possible for its class and usage.

4 The ROUNDED option means that the result is to be rounded if necessary before being stored in the result field. (See Chapter 9 for the exact rules of rounding.)

5 If the SIZE ERROR option is used and the result, rounded if specified, is too large for the result field, the result field is left as it was before the SUBTRACT operation and the imperative statement specified in the SIZE ERROR option is obeyed. If the SIZE ERROR option is not used and a size error condition occurs, the value of the result field is undefined. (See Chapter 9 for the rules of what constitutes a size error condition.)

6 If format 1 is used, the values of all the literals and identifiers preceding the word FROM are added together, the total is subtracted from the value of identifier-m and the result is stored as the new value of identifier-m.

7 If format 2 is used, the values of all the literals and identifiers preceding the word FROM are added together, the total is subtracted from literal-m or identifier-m and the result of the subtraction stored as the new value of identifier-n.

8 If format 3 is used, data items in identifier-1 are subtracted from and stored in corresponding data items in identifier-2 (see also page 67, *The CORRESPONDING option*).

9 Detailed rules of arithmetic, including those dealing with the effect of performing arithmetic on fields with different usages, are given in Chapter 9.

FUNCTION The WRITE statement releases a logical record for an output file. It can also be used for line spacing on a line printer. For direct access files, the WRITE statement also allows the performance of a specified imperative statement if the contents of the associated ACTUAL KEY field are invalid.

FORMAT 1

```
WRITE record-name [FROM identifier-1]
[ [ BEFORE ]                  [ identifier-2 LINES ] ]
[ [        ] ADVANCING        [ integer LINES      ] ]
[ [ AFTER  ]                  [ mnemonic-name      ] ]
[                             [ hardware-name      ]   ]
```

FORMAT 2 WRITE record-name [FROM identifier-1] INVALID KEY imperative-statement

NOTES

1 An OPEN statement for a file must be executed prior to executing the first WRITE statement for that file.

2 The record name must be the name of a record defined in the file section of the Data Division. The effect of a WRITE statement of either format is to copy on to the external medium associated with the file to which the record belongs the information stored in its record area. After the WRITE is executed, the record is no longer available.

3 Format 2 is used only for processing direct access files.

4 If the physical block held on the external medium holds more than one record, the programmer controls its size by the BLOCK CONTAINS clause (see page 37). WRITE statements then cause as many records as possible to be batched to produce blocks up to the size specified.

5 If the FROM option is used in either format, the effect is as if the statement

MOVE identifier-1 TO record-name

immediately preceded a WRITE statement without the FROM option. "WRITE A FROM B" has the same meaning as "MOVE B TO A; WRITE A".

6 If a file is written on to magnetic tape and a reel is filled, the program carries out the necessary actions to make the next reel available. It also carries out the dumping of information for rerun purposes if this is required. The programmer need not be aware that the file occupies more than one reel when writing the Procedure Division.

7 The ADVANCING option can only be used when the file is associated with a line printer. If BEFORE is written, the record is printed and then line spacing occurs. If AFTER is written, line spacing occurs and then the record is printed. Line spacing does not take place if the ADVANCING option is not used. If integer is specified, the printer page is advanced the number of lines equal to the value of integer. If identifier-2 is specified, the printer page is advanced the number of lines contained in identifier-2 which must identify a numeric, elementary, unedited, integral field. If mnemonic-name is specified, it must be a name given to one of the hardware names CHANNEL-n (where n is an integer in the range 1 to 7, or in the case of the 2401 Printer, 1 to 5) which refer to the channels on the tape loop used to control the line spacing of the printer. The printer page and the tape loop advance until the printing line coincides with a hole punched in the column designated in the paper tape loop. (See page 24 for mnemonic names.) If hardware-name is specified, it must be CHANNEL-n with the same meaning as above. The mnemonic-name option is provided so that hardware names which may be peculiar to 1900 COBOL need not appear in the Procedure Division.

8 Format 2 of the WRITE statement must be used

(a) for direct access files whose access mode is RANDOM and organization is DIRECT;

(b) for direct access files whose access mode is RANDOM and organization is INDEXED when it is required to access specific records.

9 If a file's access mode is SEQUENTIAL, the following two cases can occur:

(a) If the file has been opened for output, a WRITE statement places the specified record in the next physical position of the file. Such a file is always stored serially as far as the direct access system is concerned. To create a file stored in key-sequence order the Sequential File Loader program must be used.

(b) If the file has been opened for input-output, a WRITE statement must be preceded by a READ statement for the file. The WRITE statement causes the specified record to be inserted in the position immediately preceding the last record read. In the case of a file stored serially, this insertion will be carried out only if the current bucket is large enough to hold the record. If it is not, an area overflow exception condition will arise.

The insertion of a record "before the last record read" is illustrated by the following example. If the statement WRITE A followed the statement READ C, the positions of the records on the file would be AC. Another WRITE statement, for example, WRITE B, would cause the specified record to be inserted before the last record read, which in this case is still C. The positions of the records on the file would therefore be ABC.

10 A WRITE statement can only be applied to a file whose access mode is RANDOM when it has been opened for input-output.

11 If a file's access mode is RANDOM and its organization is INDEXED, the following two cases can occur:

(a) If a WRITE statement is preceded by a READ statement without an intervening SEEK statement, the record specified in the WRITE statement is inserted before the last record read.

(b) Otherwise the record specified in the WRITE statement is inserted in key-sequence order in the bucket determined by the SEEK statement and in the position in the bucket determined by the SYMBOLIC KEY field.

12 If a file's access mode is RANDOM and its organization is DIRECT, a WRITE statement must be preceded by the appropriate address generation procedure. It may be preceded by a SEEK statement with or without a READ statement. Such a WRITE statement inserts the specified record at the end of the bucket indicated by the ACTUAL KEY field.

13 The INVALID KEY procedure is executed when the logical bucket number of the required record lies outside the file limits specified when allocating the file. The logical bucket number of the required record is either indicated by the ACTUAL KEY field or determined by the execution of a SEEK statement.

Chapter 7 Program control and segmentation

A COBOL program is translated by the compiler into a sequence of 1900 series instructions which are to be held in the computer store. When the program is obeyed, it starts at a point corresponding to the first statement of the Procedure Division. The instructions corresponding to the COBOL statements are obeyed in the order in which the statements are written in the Procedure Division except when statements are met which change this order (see ALTER page 75, GO TO page 88 and PERFORM page 94).

The programmer can state the amount of storage he wishes to assign to the translated program by means of the MEMORY SIZE clause (see page 23). This storage is used for all the internal storage requirements of the program, that is, instructions, literals, constants and data areas. The compiler prints a message stating the amount of storage required for the generated program. If the program requires more internal storage than the programmer wishes to allocate to it or has available, he can arrange for some of the instructions to be held on backing store and called into core store only when required. Such a program is said to be segmented, that is, it consists of segments.

A segment is a portion of the COBOL source program comprising one or more sections. It can be either permanent or an overlay. If it is permanent, its object code is always in store at run-time; if an overlay, it is called into the core store at run-time when required and can be over-written by the next overlay segment called. COBOL subroutines cannot be segmented: i.e. irrespective of the values of the section priority numbers, and the segment limit, if present, the COBOL subroutine is compiled as a single segment. COBOL subroutines may, however, be included in overlays of a master program: the method of doing this is explained in Chapter 17.

The Procedure Division of the segmented program must be written as a series of sections, that is, it must not contain paragraphs which are not within a section.

Segmentation causes the area of core store devoted to program instructions to be divided into two areas:

1 a permanent area for instructions which remain throughout the running of the program,

2 an overlay area large enough to hold the largest overlay segment.

Segmentation also causes the program instructions to be divided into:

1 those which will occupy the permanent area,

2 two or more segments of instructions which are brought into the overlay area one at a time as required.

The program is divided into segments by assigning a priority number to each section. Sections with the same priority number belong to the same segment. The priority number of a section is assigned to each section when writing the section heading which has the following format:

section-name SECTION [priority-number]

A priority number must be an integer in the range 0 to 99 inclusive. If no priority number is given in the section heading, the section's priority number is 0.

The segments which occupy the permanent area of storage are those whose priority number is less than that written in the SEGMENT-LIMIT clause in the Environment Division (see page 23). The format of this clause is as follows:

SEGMENT-LIMIT IS priority-number

If this clause is not specified in the Environment Division sections with priority numbers up to and including 49 occupy the permanent area of storage, that is, the segment limit is 50 if it is not given explicitly.

Permanent sections without priority number, or with priority numbers up to but not including the segment limit, or up to but not including 50 if no segment limit is specified, are held in the permanent area of store. Such sections remain unchanged unless explicitly changed, for example, by an ALTER statement.

Overlay sections with priority numbers from and including the segment limit and up to but not including 50 are known as "quasi-permanent" segments. If a GO TO statement is ALTERed in such a section, it remains ALTERed when next obeyed, whether or not the section has had to be read into store again (see page 142).

Overlay sections with priority numbers from and including 50 and up to and including 99 are copied into store each time they are called for. Whenever such sections are called for, they are copied into core store in the form they had at the beginning of the program; that is, if such a segment is changed in any way whilst in store, the changes are not incorporated when it is called into core store again.

The order in which sections are written in the source program must be such that:

1 the overlay sections (priority numbers not less than the segment limit) must come first in ascending order of priority number;

2 the permanent sections come last in any order.

The effect of segmentation on the use of GO TO and PERFORM statements is as follows:

1 A GO TO Statement or a GO TO .... DEPENDING statement in any segment can refer to a section or paragraph in any segment. If the procedure name referred to is in a segment not already in core store, the segment is called in.

   A GO TO statement in a section whose priority number is greater than 49 must not be referred to by an ALTER statement in a section with a different priority number.

2 A PERFORM statement can have within its range any number of segments of any priority number.

Chapter 8 The COBOL library

The COBOL library contains text which is available to a source program at compile time. The effect of the compilation of library text is the same as if the text were actually written as part of the source program.

Text is copied from the COBOL library by means of the COPY statement. This may be written as follows:

1 After any paragraph name (but not section name) in the Environment Division;

2 After an FD, 01 or 77 level in the Data Division;

3 After a section or paragraph name in the Procedure Division.

The format of the COPY statement is as follows:

COPY library name [REPLACING word-1 BY
word-2 [word-3 BY word-4] . . .]

where library-name is composed according to the rules for COBOL data names (see page 15 of the 1900 COBOL manual), and word-n is any valid COBOL word that is not a reserved word, that is it may be a data-name, procedure-name or literal. word-1, word-3 and so on are in the library entries and are to be replaced, in the output source, by word-2, word-4 and so on.

NOTES

1 No other statement or clause may appear in the same paragraph as the COPY statement, and the " library-name " in the statement must either be on the same line as the paragraph header, or on the next line.

2 None of the words in a NOTE statement in a library entry can be REPLACEd, nor can punctuation symbols, including brackets (which for this purpose are COBOL reserved words).

3 A library entry must not itself contain a COPY statement.

4 Section headers should be held in the library in the form

FRED SECTION p .

where p is a data-name or literal.

Source program statements COPYing FRED must then be of the form

(name) SECTION . COPY FRED REPLACING p BY 0 .

if the section in the output COBOL program is to be permanent, or

(name) SECTION n . COPY FRED REPLACING p BY n .

if the section is to be overlay.

Note, however, that if the variable p occurs elsewhere in the procedural statements in FRED section, these occurrences also will be replaced by 0 or n. It is therefore advisable, in making up the COBOL library, to choose a *unique* data name or literal as p.

5 The Identification columns (73 to 80) should not be used in library entry lines, and should not be used for "comments" in source program lines containing COPY statements.

6 Other rules for the composition of COBOL library entries may be found in the User Notice on the COBOL library maintenance routine XE99.

EXAMPLES

1 *A COBOL library*

```
000000    ENVIRONMENT DIVISION
000000    ADEF
000100    OBJECT-COMPUTER. ICL–1903 MEMORY 16000 WORDS.
000000    FLOP
000100    OBJECT-COMPUTER. ICL–1903
000200            MEMORY 1000 WORDS
000300            ASSIGN OBJECT-PROGRAM PAPER-READER.
000000    FRED
000100    SOURCE-COMPUTER. ICL–1903.
000000    GROT
000100    SPECIAL-NAMES. CARD-READER 1 IS CR
000200            PRINTER 1 IS PR.
000000    JAM
000100    I-O- CONTROL. SAME RECORD AREA INFO OUTAB.
000000    PERIPH
000100    FILE-CONTROL. SELECT INFO ASSIGN CARD-READER 1.
000200            SELECT INTAB ASSIGN CARD-READER 2.
000300            SELECT OUT-TAB ASSIGN PRINTER 1.
000000    DATA DIVISION
000000    ALPHA
000100    FD ALPHA FILE 3 RECORDS RECORD 9
000200            CHARACTERS DATA RECORDS V - AREAS.
000300    01 V-AREAS
000400            02 AREAF OCCURS 3 TIMES PICTURE 9(6) DISPLAY.
000000    BETA
000100    FD BETA FILE 1 RECORDS RECORD 100 CHARACTERS
000200            DATA RECORDS FIRREC.
000300    01 FIRREC.
000400            02 YEAR PICTURE 9999.
000500            02 POPN OCCURS 3 TIMES PICTURE 9 (6) DISPLAY.
000000    DELTA
000100    01 DELTA.
000200            02 DATES PICTURE A(4) VALUE IS "YEAR".
000300            02 NAME-1 PICTURE A(6) VALUE IS "LANARK".
000400            02 NAME-2 PICTURE A(6) VALUE IS "ARGYLL".
000500            02 NAME-3 PICTURE A(6) VALUE IS "      AYR"
000000    FINAL
000100    01 FINAL PICTURE 99 COMP SYNC RIGHT VALUE 1.
```

```
000000  OMEGA.
000100  01 OMEGA.
000200          02 AREAN OCCURS 3 TIMES PICTURE 9 (6) DISPLAY.
000000  PROCEDURE DIVISION
000000  LAST
000100  LAST SECTION  PRI-NO.
000200  ONE.
000300          READ INPUT-INFO AT END GO TO FFIN.
000400  HERE.
000500          MOVE AREAF (N) TO AREAN (N)
000600          ADD 1 TO N. IF N NOT EQUAL TO 4 GO TO JACK.
000700          OPEN OUTPUT OUT-TABLE.
000800          MOVE HEADING-LINE TO HEADER-LINE.
000900          WRITE HEADER-LINE.
000000  NEXON
000100  NEXON. MOVE AREAN (N) TO QUOTIENT.
000200          DIVIDE QUOTIENT INTO POPN(N) GIVING AVPOPN(N)
000300          ROUNDED.  MOVE X TO DATE.
000400          ADD 1 to N, ADD 1 TO X.
000500          IF N NOT EQUAL TO 4 GO TO FOURTH.
000600          WRITE OUTL.  GO TO FOURTH.
000000  SOLO
000100  SOLO.
000200          OPEN INPUT IN-TABLE, OUTPUT OUT-TABLE.
000300          MOVE 1 TO N, MOVE 1960 TO X.
000400          READ IN-TABLE AT END GO TO FFIN.
****
```

2 *A COBOL input source*

```
000010  IDENTIFICATION DIVISION.
000020  PROGRAM-ID.
000030          POP100.
000040  ENVIRONMENT DIVISION.
000050  CONFIGURATION SECTION.
000060  SOURCE-COMPUTER. COPY FRED.
000070  OBJECT-COMPUTER. COPY FLOP.
000090  SPECIAL-NAMES.
000100          COPY GROT.
000110  INPUT-OUTPUT SECTION.
000120  FILE-CONTROL. COPY PERIPH
000130          REPLACING INFO BY INPUT-INFO, INTAB BY IN-TABLE,
```

```
000135          OUT-TAB BY OUT-TABLE.
000140  I-O-CONTROL. COPY JAM
000150          REPLACING OUTAB BY OUT-TABLE INFO BY IN-TABLE.
000160  DATA DIVISION.
000170  FILE SECTION.
000180  FD INPUT-INFO COPY ALPHA
000190          REPLACING 3 BY 1, V-AREAS BY AR.
000200  FD IN-TABLE COPY BETA
000210          REPLACING 100 BY 22, FIRREC BY MAIN-LINE.
000240  WORKING-STORAGE SECTION.
000250  01 HEADING-LINE COPY DELTA
000260          REPLACING DATES BY YEARS.
000270  01 RUNNING-LINE COPY OMEGA.
000280  01 N COPY FINAL.
000290  01 X PICTURE 9999 COMP SYNC RIGHT
000300          VALUE IS 1960.
000310  77 DIVISOR PICTURE 9 (6) COMP SYNC RIGHT.
000320  PROCEDURE DIVISION.
000325  HK SECTION.
000330  START.
000340          OPEN INPUT INPUT-INFO.
000350  FIRRST SECTION. COPY LAST
000360          REPLACING JACK BY HERE, HEADER-LINE
000370          BY HDDR-LINE, PRI-NO BY 0.
000375  FOURTH. COPY SOLO.
000380  SECOND SECTION.
000390  THIRD.
000400          COPY NEXON
000410          REPLACING QUOTIENT BY DIVISOR FOURTH BY THIRD
000420          OUTL BY OUT-LINE.
000430  FFIN.
000440          CLOSE INPUT-INFO, IN-TABLE, OUT-TABLE
000450          STOP RUN.
```

3 *The same program output by the COPY Phase*

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID.
000300          POP100.
000400  ENVIRONMENT DIVISION.
000500  CONFIGURATION SECTION.
000600  SOURCE-COMPUTER. ICL–1903.
```

```
000700  OBJECT-COMPUTER. ICL-1903
000800          MEMORY 1000 WORDS
000900          ASSIGN OBJECT-PROGRAM PAPER-READER.
001000  SPECIAL-NAMES. CARD-READER 1 IS CR
001100          PRINTER 1 IS PR.
001200  INPUT-OUTPUT SECTION.
001300  FILE-CONTROL. SELECT INPUT-INFO ASSIGN CARD-READER 1.
001400          SELECT IN-TABLE ASSIGN CARD-READER 2.
001500          SELECT OUT-TABLE ASSIGN PRINTER 1.
001600  I-O-CONTROL. SAME RECORD AREA IN-TABLE OUT-TABLE.
001700  DATA DIVISION.
001800  FILE SECTION.
001900  FD INPUT-INFO FILE 1 RECORDS RECORD 9 CHARACTERS
002000          DATA RECORDS AR.
002100  01 AR.
002200          02 AREAF OCCURS 1 TIMES PICTURE 9(6) DISPLAY.
002300  FD IN-TABLE FILE 1 RECORDS RECORD 22 CHARACTERS
002400          DATA RECORDS MAIN-LINE.
002500  01 MAIN-LINE.
002600          02 YEAR PICTURE 9999.
002700          02 POPN OCCURS 3 TIMES PICTURE 9 (6) DISPLAY.
002800  WORKING-STORAGE SECTION.
002900  01 HEADING-LINE.
003000          02 YEARS PICTURE A(4) VALUE IS "YEAR".
003100          02 NAME-1 PICTURE A(6) VALUE IS "LANARK".
003200          02 NAME-2 PICTURE A(6) VALUE IS "ARGYLL".
003300          02 NAME-3 PICTURE A(6) VALUE IS "      AYR".
003400  01 RUNNING-LINE.
003500          02 AREAN OCCURS 3 TIMES PICTURE 9(6) DISPLAY.
003600  01 N PICTURE 99 COMP SYNC RIGHT VALUE 1.
003700  01 X PICTURE 9999 COMP SYNC RIGHT
003800          VALUE IS 1960.
004000  77 DIVISOR PICTURE 9(6) COMP SYNC RIGHT.
004100  PROCEDURE DIVISION.
004200  IIK SECTION.
004300  START.
004400          OPEN INPUT INPUT-INFO.
004500  FIRRST SECTION 0
004600  ONE.
004700          READ INPUT-INFO AT END GO TO FFIN.
```

```
004800 HERE.
004900         MOVE AREA (N) TO AREAN (N).
005000         ADD 1 TO N. IF N NOT EQUAL TO 4 GO TO HERE.
005100         OPEN OUTPUT OUT-TABLE.
005200         MOVE HEADING-LINE TO HDDR-LINE.
005300         WRITE HDDR-LINE.
005400 FOURTH.
005500         OPEN INPUT IN-TABLE, OUTPUT OUT-TABLE.
005600         MOVE 1 TO N, MOVE 1960 TO X.
005700         READ IN-TABLE AT END GO TO FFIN.
005800 SECOND SECTION.
005900 THIRD.  MOVE AREAN (N) TO DIVISOR.
006000         DIVIDE DIVISOR INTO POPN(N) GIVING AVPOPN(N)
006100         ROUNDED. MOVE X TO DATE.
006200         ADD 1 TO N, ADD 1 TO X.
006300         IF N NOT EQUAL TO 4 GO TO THIRD.
006400         WRITE OUT-LINE. GO TO THIRD.
006500 FFIN.
006600         CLOSE INPUT-INFO, IN-TABLE, OUT-TABLE.
006700         STOP RUN.
* * * *
```

Chapter 9 Rules of arithmetic

INTRODUCTION

There are four arithmetic verbs, namely, ADD, SUBTRACT, MULTIPLY and DIVIDE. The formats of statements involving them are given on pages 74, 102, 91 and 82 respectively of this manual.

MOVE is also considered an arithmetic verb in the cases where one of its operands is a decimal field or literal and the other is a binary field since arithmetic conversion occurs in these cases (see page 114 below).

Definitions

"The point position" of any quantity is considered to be positive (+N places) if the quantity has point left N places and negative (-N places) if it has point right N places.

"The result" of a computation is simply the numeric result of the computation at run-time.

"The receiving field" is the field in which the result is finally stored.

Basic concepts

The following comments are intended only as a general outline. Modifications and variations due to the individual verbs and the clauses of their operands will be considered later in greater detail. For purposes of explanation it is assumed that the result and the receiving field have the same class.

1 ROUNDING If after decimal point alignment the number of places in the fraction of the result of and arithmetic operation is greater than the number of places provided for the fraction in the receiving field, truncation to the size of the receiving field takes place. However, if ROUNDED is specified, the absolute value of the receiving field is increased by 1 whenever the most significant digit of the excess is greater than or equal to half the radix of that place. When integer positions in a receiving field are represented by the character P in the picture for that receiving field, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated. If no truncation could possibly take place and ROUNDED had been specified, the compiler outputs the message "ROUNDING CLAUSE POINTLESS".

2 SIZE ERROR If after decimal point alignment the value of a result exceeds the largest value that can be contained in its specified receiving field, a size error condition exists. When such a size error condition occurs, the subsequent action of the compiler depends on whether or not the SIZE ERROR option has been specified.

If SIZE ERROR has not been specified and a size error condition occurs the contents of the specified receiving field will be unpredictable.

If SIZE ERROR has been specified and a size error condition occurs the contents of the receiving field will remain unchanged and the SIZE ERROR clause will be obeyed.

3 THE COMPOSITE The composite may be considered as the result of a computation in which all the operands hold their maximum possible values. If such a result is too large for the compiler to deal with correctly, the message "COMPOSITE TOO BIG" is output. This message indicates a caution except when it is explicitly stated that it indicates an error. The action taken by the compiler on encountering this condition depends on the particular case.

In general, if a "COMPOSITE TOO BIG" message has been signalled, run-time results which exceed the maximum permitted composite size will not be detected by a SIZE ERROR test. Results of such computations may therefore be unpredictable even when a SIZE ERROR condition has been specified.

4 NEGATIVE RESULTS

(a) Rounding is always performed on the modulus of the result, i.e. on the absolute value of the result, ignoring whether it is signed or not, e.g. -3.5 is rounded to -4 and not to -3.

(b) If the result of a computation is negative and the receiving field is unsigned, the modulus or absolute value of the result will be placed in the receiving field and the sign will be ignored, e.g. -5185 would be stored as 5185.

Types of computation

In this section the concepts described above are developed in greater detail. For this purpose four types of computation are distinguished according to the class of fields occurring in them and the rules for each COBOL verb are given for each type. Much of this material, in particular that related to mixed computations, is of rather specialized interest and users will in general wish to refer to it only when they get an unexpected error message from the compiler.

The four types of computation are as follows:

1 DECIMAL in which each of its operands is either a decimal field or a numeric literal.

2 STERLING in which at least one of its operands is a sterling field and each of its operands is either a sterling field, a decimal field or a numeric literal.

3 BINARY in which each of its operands is a binary field or an integral numeric literal.

4 MIXED in which at least one of its operands is a binary field and at least one other is either a decimal field, a sterling field or a non-integral numeric literal.

Note: In all the above definitions, the receiving field is considered as one of the operands.

DECIMAL COMPUTATIONS

The maximum permissible size of any operand of a decimal computation except the receiving field is 12 decimal digits. If any operand exceeds the maximum the compiler will output the message "ILLEGAL ARITHMETIC OPERAND" and will ignore the operand.

ADD and SUBTRACT

The composite is the sum of the moduli or absolute values of the maximum possible values of all the addends or subtrahends and must not exceed 12 decimal digits in length. If it does the compiler will output the message "COMPOSITE TOO BIG" and will assume that the run-time result of the computation will never be longer than 12 decimal digits. If this assumption is not true, the results of the computation will be unpredictable.

MULTIPLY

The composite is within range on decimal multiplication if:

(the number of places before the point in the multiplicand)
+ (the number of places before the point in the multiplier)
+ (the number of places after the point in the receiving field)
(+1 if ROUNDED is specified)

is less than 13.

If the composite is out of range the compiler will output the message "COMPOSITE TOO BIG" and assume that the run-time result will nevertheless satisfy the above conditions. If it does not the results of the computation will be unpredictable.

DIVIDE

The composite is within range if:

(the number of places before the point of the dividend)
+ (the number of places after the point of the divisor)
+ (the number of places after the point of the receiving field)
(+1 if ROUNDED is specified)

is less than 13.

If the composite is not within range, the compiler will print the message "COMPOSITE TOO BIG". In this case the result will lose N decimal places off the least significant end, where N is the number of digits by which the composite exceeds 12, if ROUNDED was not specified or (N-1) digits if ROUNDED was specified. In the latter case, moreover, rounding will not take place.

STERLING COMPUTATIONS

The most important thing to appreciate about the way the compiler handles sterling computations is that all sterling arithmetic is really decimal arithmetic. Any sterling field before being operated on is converted to pence and the resulting decimal quantities are operated on in exactly the same way as ordinary decimal fields. Since the compiler treats sterling computations in this way, it is unnecessary to give such detailed rules for sterling computations as have been given for decimal computations.

In general, to detect possible out of range composites and size errors, all the user has to do is to find the maximum value and the equivalent point position of the fields considered as pence and then calculate the composite size and point position as if the computation were a purely decimal one. As mentioned earlier, no decimal arithmetic field can be longer than 12 decimal digits, Similarly, the maximum equivalent value in decimal pence of a sterling field must not be longer than 12 decimal digits.

If the sterling fields are integral or have point left, it is fairly obvious how to convert then to decimal, i.e. multiply the maximum contents of the pounds places by 240, the maximum contents of the shillings places by 12 and add the results to the maximum of the pence and decimal of pence places.

The less common point right cases are more difficult since the point positions of the equivalent decimal quantities are not generally the same as those of the original sterling quantities. For example, 99PPP USAGE DISPLAY-1 represents a sterling field in the range £0 to £99. Given any value N in the field the equivalent number of pence is N × 240. Instead of holding this decimal equivalent as point 3 right the compiler holds it as point 1 right since the decimal equivalent of a point 3 right sterling field will always be a multiple of 10. If the decimal equivalent were represented as point 3 right, only multiples of 1000 could be held and a value such as £56 (decimal equivalent 13440) could not be held accurately in its decimal equivalent.

Again, for example, consider a field specified as 999PPP USAGE DISPLAY-2 which has two positions of pounds and one of tens of shillings. The maximum decimal equivalent is therefore (99 × 240) + (1 × 120) i.e. 23880. It is clear that the decimal equivalent of any value contained in the field must be a multiple of 10. Because this is so the compiler regards the maximum value as 2388 point 1 right and the user too should regard it as such when calculating composites and size error possibilities.

Therefore, when a point right sterling field is converted to its decimal equivalent, the point position of the field is very often changed too as can be seen from the following table:

Sterling point	Point position of decimal equivalent if DISPLAY-1	if DISPLAY-2
1 Right	0	0
2 Right	1 Right	0
3 Right	1 Right	1 Right
4 Right	2 Right	1 Right
.	.	.
.	.	.
.	.	.
(4 + N) Right	(2 + N) Right	(1 + N) Right

where N is the difference between 4 and the sterling point number whose decimal equivalent point position is required.

As has been seen the total size of the equivalent maximum decimal quantity must not be more than 12 digits. The rules given above for the calculation of maximum equivalent decimal quantities are sufficient for the user to decide whether or not his sterling fields are legal arithmetic operands. However, since in practice working out the equivalent decimal could be rather cumbersome, the

following simple general rule can be used for most cases: *a sterling field is a legal arithmetic operand if the number of places of pounds + the number of places of decimals of pence is less than 10.*

Rounding with sterling receiving fields

If the receiving field has point left or is integral the rounding rule is the same as for decimal computations i.e. 1 is added to the receiving field after truncation if the most significant digit lost by truncation is greater than or equal to 5.

EXAMPLE: ADD 3.4, A GIVING B ROUNDED

where A has PICTURE 9(3)V9 USAGE DISPLAY-1 and current contents 19;1 (i.e. 19/11.1d) and B has PICTURE 9(4) USAGE DISPLAY-1.

This would give 1003 (£1 – 0 – 3) as the result in B.

If the literal had been 3.3, the result would have been £1 – 0 – 2.

It the receiving field has point right, rounding is performed by adding one half of the radix of the most significant place lost by truncation to the result held as a decimal equivalent.

EXAMPLE: ADD 6.9, A GIVING B ROUNDED

Where A has PICTURE 9(3)V9 USAGE DISPLAY-1 and current contents 19;1 (i.e. 19/11.1d) and B has PICTURE 9(3)P USAGE DISPLAY-1. This would give a result in B of 101 (i.e. £1 – 1 – 0). Had the literal been 6.8 the result would have been £1 – 0 – 0.

If the decimal point or equivalent decimal point of the result as well as the sterling point of the receiving field is point right, a special case of composite error can arise which is flagged by the compiler with the message "POSSIBLE ROUNDING OVERFLOW". This occurs if:

(the size in digits of the composite)
- (the decimal point of the composite)
+ (the equivalent decimal point of the receiving field)

is greater than 12.

BINARY COMPUTATIONS

The maximum size permitted for any binary field used in arithmetic statements is 46 binary digits excluding the sign.

ADD and SUBTRACT

The composite is the sum of the moduli or absolute values of the maximum possible values of all the addends or subtrahends and is in range if it is less than 47 binary digits in length.

If the composite is not within range the compiler will output the message "COMPOSITE TOO BIG". However, should all the run-time results of the computation not exceed 46 binary digits they will be processed correctly. If some run-time results do exceed this maximum the contents of the receiving field will be unpredictable.

MULTIPLY

The composite is within range if:

(the number of places before the point in the multiplicand)
+ (the number of places before the point in the multiplier)
+ (the number of places after the point in the receiving field)
(+1 if rounding is specified)

is less than 47.

If the composite is out of range the compiler will print the message "COMPOSITE TOO BIG" and assume that the run-time results will nevertheless satisfy the above condition. If it does not, the result of the computation will be unpredictable.

DIVIDE

The composite is within range if:

(the number of places before the point of the dividend)
+ (the number of places after the point of the divisor)
+ (the number of places after the point of the receiving field)
(+1 if rounding is specified)

is less than 47.

If the composite is out of range the compiler will output the message "COMPOSITE TOO BIG". All run-time results will then lose N binary places off the least significant end, where N is the number of digits by which the composite exceeds 46, if rounding had not been specified. If rounding had been specified it will not take place and the results will lose (N-1) binary places.

MIXED COMPUTATIONS

A computation is considered as mixed if at least one of the operands is binary and at least one other is either a decimal field, a sterling field or a non-integral numeric literal.

Numeric MOVE

A MOVE of a binary to a decimal field or vice versa is considered an arithmetic operation since arithmetic conversion takes place. The decimal field may not exceed 12 decimal digits in length.

1 BINARY TO DECIMAL

The result of a MOVE is out of range on conversion if:

$DP + (BS - BP) \log_{10}2$ is greater than 12, where:

DP is the point position of the decimal field;

BS is the size in digits of the binary field;

BP is the point position of the binary field.

2 DECIMAL TO BINARY

The result of a MOVE is out of range on conversion if:

$BP + (DS - DP) \log_{2}10$ is greater than 46, where:

BP is the point position of the binary field;

DS is the size in digits of the decimal field;

DP is the point position of the decimal field.

Note: For the purposes of these calculations .3010 and 3.323 can be taken for the values of $\log_{10}2$ and $\log_{2}10$ respectively.

If the result of a mixed MOVE is out of range on conversion, the compiler assumes that the run-time value of the source field is small enough for the result to be within range.

The value of the point position of a binary field in a mixed MOVE must be in the range -46 to +46. Similarly, the point position of a decimal field must be in the range -12 to +12.

ADD and SUBTRACT

If in the case of ADD and SUBTRACT...GIVING only the receiving field has a different class (i.e. is binary whilst all the operands are decimal or sterling or is decimal or sterling whilst all the operands are binary) the composite rules are the same as for unmixed additions and subtractions. The result is then moved to the receiving filed in accordance with the rules for numeric move with the exception that if it has more digits to the left of the decimal point than the receiving field the normal rules for size error conditions are followed and no truncation of the most significant digits takes place.

In other cases the binary and decimal fields are added separately to form subtotals. All literals are added into the decimal subtotal. If composite overflow occurs as a result of the addition of these subtotals, the compiler action is the same as for unmixed cases.

The binary subtotal is then converted to a decimal number point D, where D is either D_1 (the decimal point of the decimal subtotal) or $B_1 \log_{10}2$ (where B_1 is the binary point of the binary subtotal), depending on whichever is greater, according to the rules explained above for the conversion of decimal to binary fields.

MULTIPLY

In mixed computations the message "COMPOSITE TOO BIG" may indicate an error rather than a caution. This happens when the result of the computation would exceed the 92 binary places (4 words) of working storage area assigned to it. In this case the computation is not performed and the contents of the receiving field remain unchanged. It occurs if $S_1 + S_2 + B_1 + D_1$ rounded to the next integer is greater than 92, where:

S_1 is the size in binary digits of the multiplicand

S_2 is the size in binary digits of the multiplier

B_1 is equal to either

(a) the modulus of B if B is negative, i.e. point right or

(b) zero if B is positive i.e. point left.

(B is the binary point position of the product which is the sum of the binary point position of the multiplicands minus the binary point position of the receiving field).

D is equal to either

(a) the modulus of D is D is negative i.e. point right or

(b) zero if D is positive i.e. point left.

(D is the decimal point position of the product which is the sum of the decimal point position of the multiplicands minus the decimal point position of the receiving field).

Apart from this case, the rules for mixed multiplication with a binary receiving field are the same as those for binary multiplications and those for mixed multiplication with a decimal receiving field are the same as those for a decimal multiplication with the exceptions that:

(i) if the receiving field is binary and the multiplicand or multiplier is also binary "the number of places before the point in the multiplicand/multiplier" (see page 112) has its normal meaning; but if the multiplicand or multiplier is decimal, it means (the number of decimal places before the point in the multiplicand or multiplier) $\times \log_2 10$;

(ii) similarly, if the receiving field is decimal, the number of places before the point in a binary multiplicand or multiplier means (the number of binary places before the point in the multiplicand or multiplier) $\times \log_{10}2$.

DIVIDE

The decimal and binary points of the quotient, referred to as D and B respectively are calculated as follows. D is equal to (the decimal point of the dividend) – (the decimal point of the divisor) – (the decimal point of the receiving field). B is equal to (the binary point of the dividend) – (the binary point of the divisor) – (the binary point of the receiving field). In calculating D and B the binary point of a decimal field and the decimal point of a binary field are taken to be zero. If rounding is specified for a decimal receiving field, D is 1 less than indicated; similarly, if for a binary field, B is 1 less than indicated.

The size of the quotient in binary digits, referred to as N, is then calculated on the basis that:

$$N = (\text{Bit size of dividend}) - B - D \times \log_2 10.$$

If the receiving field is binary and N is greater than 46 the composite is out of range and the loss of accuracy is (N-46) binary places. If the receiving field is decimal and N is greater than 41 the composite is out of range and the loss of accuracy is (N-41) $\log_{10} 2$ decimal places.

Chapter 10 Moving data in the store

Data may be moved from one area to another. However, due to the different forms specifiable for fields in 1900 COBOL, the final contents of the receiving field cannot be determined without reference to the USAGE and PICTURE of both the source and receiving fields. The rules for determining the results of MOVEs among the various types of data fields are given in this chapter. Other COBOL statements which have a similar effect refer to this chapter where necessary.

MOVING UNEDITED ELEMENTARY FIELDS TO FIELDS OF THE SAME CLASS AND USAGE

1 ALPHABETIC to ALPHABETIC

The characters of the source field are transferred into the corresponding positions of the receiving field, starting with the leftmost. If the source field is shorter than the receiving field, the unfilled characters in the receiving field are space-filled. If the source field is longer than the receiving field, the excess characters in the source field are ignored. If the receiving field is described as JUSTIFIED RIGHT, the source and receiving fields are aligned such that the rightmost characters correspond and any space filling or transaction takes place on the left.

2 ALPHANUMERIC to ALPHANUMERIC

The same rules apply as for alphabetic to alphabetic.

3 DISPLAY NUMERIC to DISPLAY NUMERIC

The decimal points of the source and receiving fields are aligned, that is, the character immediately before the point in the source field is aligned with the character immediately before the point in the receiving field. There may then be more or fewer digits before or after the decimal point in the source field than in this field. If there are more, only those digits that align with the characters in the receiving field are transferred and excess digits to the left or right are truncated. If there are fewer, the unfilled positions in the receiving field are set to zero.

If both the fields are signed, the sign in the leading character position of the source field is put into the leading character position of the receiving field. If the source field is signed but the receiving field is unsigned, the "overpunched" sign over the leftmost character of the source field is not transferred to the receiving field, that is, a negative number becomes positive when MOVEd from a signed to an unsigned field.

EXAMPLES

```
02   FRED   PICTURE   S99V999   DISPLAY.
02   JIM    PICTURE   S999V9    DISPLAY.
MOVE   FRED   TO   JIM.
```

If FRED equals -23.245 held as K3456, JIM becomes "234

If FRED equals +0.123 held as 00123, JIM becomes 0001

If FRED equals +0.123 held as ▽▽ 123, JIM becomes 0▽▽1

Note how in the first case the sign is transferred. All three cases illustrate zero filling on the left. The third case illustrates a tricky point. Whenever a COBOL program converts a value to DISPLAY form, leading non-significant digits are represented by zeros. However, in the case of a straight transfer as illustrated, the aligned characters are transferred as they are. The fact that the leading non-significant digits in JIM are represented by a mixture of zeros and spaces does not matter in as much as any subsequent arithmetic operations on JIM will be effected correctly and efficiently. Consider:

MOVE JIM TO FRED.

If JIM equals -456.7 held as M567, FRED becomes N6700.

If JIM equals 0.6 held as 0006, FRED becomes 00600.

If JIM equals 0.6 held as $\nabla\nabla\nabla$6, FRED becomes $\nabla\nabla$600.

The first case shows how significant digits have been lost from the left-hand end. FRED then represents -56.700. All the cases illustrate zero filling on the right.

4 DISPLAY-n to DISPLAY-n

If a DISPLAY-n field is moved to another of the same type, it obeys the rules of alignment, truncation, zero-filling and treatment of leading spaces and signs given for moving a DISPLAY NUMERIC field.

5 COMPUTATIONAL to COMPUTATIONAL

The effect of such a move is the same as a move between two numeric DISPLAY fields with the same respective Pictures as the two COMPUTATIONAL fields, that is, the decimal truncation effects, if any, are the same.

EXAMPLES

```
02  FRED  PICTURE  S99V999  COMPUTATIONAL.
02  JIM   PICTURE  S999V9   COMPUTATIONAL.
MOVE  FRED  TO  JIM.
```

If FRED held -23.456 in COMPUTATIONAL form, JIM would be set to -23.4 in COMPUTATIONAL form. Similarly, if the move were in the opposite direction and JIM held -456.7 in COMPUTATIONAL form, FRED would be set to -56.700 in COMPUTATIONAL form.

6 COMPUTATIONAL-1 to COMPUTATIONAL-1

The binary points of the source and the receiving fields are aligned, that is, the bit immediately before the point in the value of the source field is aligned with the bit immediately before the point in the receiving field. There may be more or fewer bits before or after the point in the source field than in the receiving field. If there are more, only those bits which align with the bits in the receiving field are transferred, that is, truncation of the excess bits takes place. If there are fewer, the value is transferred and unfilled bits to the right of it are zero filled. If the fields are unsigned, any unfilled bits to the left are zero filled. If, however, the fields are signed, any unfilled bits to the left are sign-filled that is, the sign of the value is propagated to the left-hand end of the field.

EXAMPLES

```
02  FRED  PICTURE  S1(5)V1  COMPUTATIONAL-1.
02  JIM   PICTURE  S111     COMPUTATIONAL-1.
MOVE  FRED  TO  JIM.
```

If FRED equals + 5 held as 0001010, JIM becomes 0101, that is, + 5.

If FRED equals - 5 held as 1110110, JIM becomes 1011, that is, - 5.

If FRED equals + 30 held as 0111100, JIM becomes 0110, that is, + 6.

If FRED equals - 30 held as 1000100, JIM becomes 1010, that is, - 6.

If FRED equals + 5.5 held as 0001011, JIM becomes 0101, that is, + 5.

If FRED equals - 5.5 held as 1110101, JIM becomes 1011, that is, - 5.

The first two examples are straightforward. The next two illustrate truncation on the left; the value is truncated and the sign bit is moved across. The last two illustrate truncation on the right. The last example illustrates that truncation is not effected by simply transferring bits 4 to 6 of FRED. This would give JIM equal to 1010, that is, -6. Truncation of the modulus is required. Therefore, to move a field like FRED to one like JIM when the fields are signed and the receiving field is held to less accuracy than the source field, the object program effectively tests if the value is negative, and, if it is, complements before truncating and then complements back.

When the move is in the opposite direction, that is, the statement is:

```
MOVE   JIM   TO   FRED.
```

If JIM equals + 5 held as 0101, FRED becomes 0001010, that is, + 5.

If JIM equals - 5 held as 1011, FRED becomes 1110110, that is, - 5

These examples illustrate zero fill on the right and sign propagation on the left. Such examples are straightforward bit transfers, that is, no complementing is necessary.

These rules also apply when the fields are more than 23 bits in length since it is the bits of the value proper that are aligned. (The 24th 'null' bit does not affect the transfer.) If a bit pattern is defined as COMPUTATIONAL-1, it is correctly moved to another such field since it is effectively an unsigned integer. If it is longer than 23 bits, the null bit is presumably part of the pattern but, since both fields are effectively unsigned integers, the null bit is transferred correctly.

MOVING UNEDITED ELEMENTARY FIELDS TO FIELDS OF DIFFERENT CLASS OR USAGE

If the source field is of a different class or usage from the receiving field, the object program must first convert the value from the source field into the form in which the receiving field is represented. The moving of the converted value into the receiving field then obeys the rules given for moving fields to fields of the same class and usage. A move which involves an arithmetic conversion is considered an arithmetic operation which affects the size limit of the source field (see Chapter 9). The following paragraphs list those moves which are permitted and the conversions involved.

1 Receiving field ALPHABETIC

Only ALPHABETIC or ALPHANUMERIC fields can be moved to an ALPHABETIC field. No conversion is required and the transfer obeys the rules for moving ALPHABETIC to ALPHABETIC fields. The character values are assumed to belong to the alphabetic set (A to Z and space). No check is made but the programmer can make one explicitly by using "If data-name ALPHABETIC..."

2 Receiving field ALPHANUMERIC

Any type of field can be moved to an ALPHANUMERIC field. ALPHABETIC, DISPLAY and DISPLAY-n fields require no conversion and are moved as if they were ALPHANUMERIC. The sign, whether separate or overpunched, is also moved.

A COMPUTATIONAL field is first converted to a DISPLAY-3 form with an equivalent Picture to that of the COMPUTATIONAL field. The resulting character string is then treated as an ALPHANUMERIC field.

A COMPUTATIONAL-1 field is first converted so that each bit including the sign bit (if any) is stored as a separate character. The resulting character string is then moved as a left-justified ALPHANUMERIC field which is truncated or space-filled on the right as necessary, unless the receiving field is described as JUSTIFIED RIGHT in which case the character string is moved right-justified with truncation or space-filling on the left as necessary.

3 Receiving field DISPLAY NUMERIC

Any class of field except ALPHABETIC can be moved to a DISPLAY NUMERIC field.

All types of NUMERIC fields are first converted to DISPLAY NUMERIC form and the converted value is then transferred to the receiving field obeying the rules for moving DISPLAY NUMERIC to DISPLAY NUMERIC.

A DISPLAY-3 field is converted to DISPLAY NUMERIC form by removing the sign-digit (if any) and inserting the correct "overpunched" sign over the leftmost character of the converted value.

Sterling fields (DISPLAY-1, -2, -4, and -5) are converted to pence in DISPLAY NUMERIC form. If signed, the correct "overpunched" sign is inserted over the leftmost character of the converted value.

EXAMPLE

```
02   FRED   PICTURE   S9999999V99   DISPLAY-5.
02   JIM    PICTURE   S9999V9      DISPLAY.
```

If FRED equals – £4/12/10.25d held as -004121025, JIM becomes J1142, that is, JIM holds -1114.2 pence in DISPLAY NUMERIC form.

A COMPUTATIONAL field is converted to DISPLAY NUMERIC form with the number of decimal digits indicated in the Picture of the COMPUTATIONAL field.

A COMPUATIONAL-1 field is converted to decimal form, accurate to the number of decimal places in the receiving field. The transfer of the converted value obeys the laws of alignment, truncation, zero-filling and signing given for moving DISPLAY NUMERIC to DISPLAY NUMERIC.

If the source field is ALPHANUMERIC, the value is transferred left-justified with truncation or zero fill on the right. The character values are assumed to belong to the correct character set.

4 Receiving field DISPLAY-n

The rules given for a DISPLAY NUMERIC receiving field apply except that the conversion is to the appropriate sterling or numeric form. When moved to sterling fields, the values of DISPLAY NUMERIC, COMPUTATIONAL or COMPUTATIONAL-1 fields are considered to represent pence and fractions of pence.

EXAMPLE

```
02   FRED   PICTURE   S9999999V99   DISPLAY-5.
02   JIM    PICTURE   S9999V9      DISPLAY.
MOVE   JIM   TO   FRED.
```

If JIM equals -1114.2 held as J1142 FRED becomes -004121020, that is, FRED represents -£4/12/10.2d in DISPLAY-5 form.

Moves which merely involve changes of sign or pence representation are not considered as arithmetic conversions for the purposes of the size rules. Conversions from sterling to numeric and vice versa, however, are considered as arithmetic conversions. If the source field is ALPHANUMERIC and the receiving field is signed with a separate sign character, the left-justified transfer means that the first character of the alphanumeric field is taken as the sign character. The character values of the ALPHANUMERIC field are assumed to be consistent with the position into which they are transferred.

5 Receiving field COMPUTATIONAL

The rules concerning the classes of fields that may be moved to a COMPUTATIONAL field and the effects of alignment, truncation zero-filling and signing are the same as those for moving to a DISPLAY-3 field with the same Picture as the COMPUTATIONAL field. (COMPUTATIONAL receiving fields are described as being similar to DISPLAY-3 rather than to DISPLAY NUMERIC because when an ALPHANUMERIC field is moved to a signed COMPUTATIONAL field, the first character is taken as a sign character.)

6 Receiving field COMPUTATIONAL-1

Any class of field except ALPHABETIC can be moved to a COMPUTATIONAL-1 field.

DISPLAY NUMERIC and COMPUTATIONAL fields are converted to COMPUTATIONAL-1 fields with the same number of binary places after the point as the receiving field. Sterling fields are similarly converted to binary pence. The rules of alignment, truncation and zero-filling of the converted value are the same as for moving COMPUTATIONAL-1 to COMPUTATIONAL-1.

ALPHANUMERIC fields are converted to bit form by taking each character modulo 2 as a bit and transferring the resultant bit string left-justified with truncation or zero-filling on the right if necessary.

EXAMPLE

```
02   FRED   PICTURE   X(4)   DISPLAY.
02   JIM    PICTURE   1(4)   COMPUTATIONAL-1.
MOVE   FRED   TO   JIM.
```

If FRED equals ABC9 held as 100001100010100011001001, JIM becomes 1011.

MOVING SIGNED VALUES TO UNSIGNED FIELDS AND VICE VERSA

If a field described as signed is moved to a field described as unsigned, the absolute value of the sending field is transferred. In the case of character fields this means that the overpunched or separate sign character is not transferred. Negative COMPUTATIONAL-1 fields are complemented before transfer. If the COMPUTATIONAL-1 field requires truncation on the right, it is complemented before truncation takes place.

If an unsigned field is moved to a signed field, its value is taken to be positive.

MOVING SYNCHRONIZED FIELDS

The fact that either the source field or the receiving field is SYNCHRONIZED does not alter the result of a move in any way. The only difference is that the compiler is sometimes able to generate code to take advantage of the fact and effect the transfer more efficiently. In particular, COMPUTATIONAL and COMPUTATIONAL-1 fields are moved more efficiently when they are described as SYNCHRONIZED RIGHT.

A source of error, however, is to consider a field such as

```
02   FACTOR   PICTURE   S9   COMPUTATIONAL   SYNCHRONIZED   RIGHT,
```

as being one word long when considering the range of numbers it can hold and the truncation of values moved to it. Although such a field occupies one word of storage, it contains only one decimal place. The fact that it is SYNCHRONIZED RIGHT should therefore be ignored when the truncation of values destined for such a field is considered.

MOVING LITERALS TO FIELDS

Numeric and non-numeric literals obey the same rules as numeric DISPLAY-3 and ALPHANUMERIC fields respectively. Hence, the types of fields to which they can be moved and the effect of such moves are the same as for these types of fields.

There are no sterling or binary literals but any given sterling or binary value can be obtained in these types of fields by moving either the corresponding numeric literal or suitably chosen alphanumeric literals. The latter effectively constitute pseudo-sterling or binary literals. For example, if it were required to move the value £2/19/6½d to a field described as

```
02   PRICE   PICTURE   S999999V9   DISPLAY-2,
```

either of the following statements would be written:

```
MOVE   714.5   TO   PRICE.
MOVE   "0219065"   TO   PRICE.
```

Using the non-numeric literal saves effecting the conversion to pence (£2/19/6½d equals 714.5 pence) but it has to be laid out properly. It also gives greater notational freedom in so far as leading zeros can be represented by spaces or zeros and, if the field has usage DISPLAY-1, tenpence and elevenpence can be represented by either convention. For example, if PRICE were DISPLAY-1, the value £2/19/10d.,

could be written as 718, "0219&" or "0219:" (trailing zeros can be omitted from non-numeric literals since they are transferred left-justified with zero fill on the right). However, care must be taken with negative values, when using the non-numeric literal, to represent the sign correctly, since the quotation mark cannot be used for an overpunched zero. For example, if it were required to move the value -£2/19/6½d to PRICE either of the following statements could be written:

```
MOVE   -714.5    to   PRICE.
MOVE   "-219065"  to   PRICE.
```

The minus sign in the non-numeric literal represents an overpunched zero. If the value were -£12/19/6½d., it would be written as a non-numeric literal as "J219065". If PRICE were DISPLAY-4 (separate sign character), it would be written as "-1219065"

Non-numeric literals can be used similarly to move bit patterns into COMPUTATIONAL-1 fields. The rules for forming literals to move to a COMPUTATIONAL field are the same as for a DISPLAY-3 field with an equivalent Picture.

Literals moved to fields described by a Symbol P must be correctly scaled. For example, if a field were described as:

```
02   AMOUNT   PICTURE   99PPP   DISPLAY
```

AMOUNT represents thousands. The literal must therefore be written as thousands for it to be aligned correctly. The following statement would therefore be incorrect since 345 would be truncated.

```
MOVE   12345   TO   AMOUNT.
```

The rules for the VALUE IS clause in the Data Division are exactly the same as the rules for moving literals to fields, except that the clause must not be associated with fields described as edited.

MOVING FIGURATIVE CONSTANTS TO FIELDS

The figurative constant ZERO or ZEROS or ZEROES can be considered as a numeric literal and all other figurative constants listed on pages 16 and 17 as alphanumeric literals whose length is such as to completely fill the receiving field.

MOVING NON-ELEMENTARY FIELDS

A field is called quasi-character if it is an exact number of characters in length, that is, its object program size is a multiple of six binary digits, and if it starts on a character boundary.

If both source and receiving fields are quasi-character non-elementary fields, a direct transfer of data is affected, left justified and with space filling or truncation on the right.

If one field is elementary and the other non-elementary, the non-elementary field is treated as an elementary alphanumeric field and the MOVE obeys the rules given on page 117 for such moves. (Editing and conversion of data from one form to another may occur.)

If a non-elementary field is not quasi-character it can only be moved to another non-elementary field. In this case fields must have the same object program size and must not be subscripted with data-name subscripts. A direct block transfer is effected.

All other moves produce the message "ILLEGAL GROUP OR RECORD MOVE" (see message 317 on page 218).

EDITING

1 Editing ALPHABETIC fields

The Picture of an edited ALPHABETIC field contains one or more occurrence of both the characters A and B. When a value is moved to such a field the characters of the value are placed in the field in the positions corresponding to A in the Picture. Spaces are put into the field in the

positions corresponding to B in the Picture. The value is moved in from the left-hand end (from the right-hand end if the receiving field is JUSTIFIED RIGHT) and, apart from the positions corresponding to B, follows the rules for moving to unedited ALPHABETIC fields (see page 117).

If an edited ALPHABETIC field is a source operand, that is, the first operand of a MOVE statement, all the characters including the editing symbols are moved.

EXAMPLES

```
02  FRED  PICTURE  AAAAA.
02  JIM   PICTURE  ABABBAAAA.
MOVE  FRED  TO  JIM.
```

If FRED equals "SMITH", JIM becomes "S␣M␣␣ITH␣".

```
MOVE  JIM  TO  FRED.
```

If JIM equals "B␣R␣␣OWNE", FRED becomes "B␣R␣␣".

2 Editing ALPHANUMERIC fields

The Picture of an edited ALPHANUMERIC field contains one or more occurrence of the character X and one or more occurrence of either or both of the characters B and 0. When a value is moved to such a field, the characters of the value are placed in the field in positions corresponding to X in the Picture. The value is moved in from the left-hand end (from the right-hand end if the receiving field is JUSTIFIED RIGHT) and, apart from the positions corresponding to B and 0, follows the rules for moving to an unedited ALPHANUMERIC field (see pages 117 and 119).

If an edited ALPHANUMERIC field is a source operand that is, the first operand of a MOVE statement, all the characters including the editing symbols are moved.

EXAMPLES

```
02  FRED  PICTURE  XXXXX.
02  JIM   PICTURE  XXXB0XX.
MOVE  FRED  TO  JIM.
```

If FRED equals "NO.23", JIM becomes "NO. ␣023".

```
MOVE  JIM  TO  FRED.
```

If JIM equals "NO.␣023", FRED becomes "NO.␣0".

3 Editing NUMERIC fields

A numeric field can be defined as edited

(a) by the use of the editing clauses or

(b) by a Picture which contains one or more occurrences of one or more of the characters or character combinations Z [*] £ $ [-] [+] CR DB [.] B 0 [,].

An edited NUMERIC field must have usage DISPLAY or DISPLAY-n. The rules for moving a value to such a field are the same as for moving to an unedited NUMERIC field with the same usage except for the differences described below. The permissible combinations of the above characters in a Picture and the rules concerning where they can appear in a Picture are given on page 51).

SIMPLE INSERTION CHARACTERS

The characters [,] B 0 [.] can appear anywhere in the Picture of an edited NUMERIC field (however, see page 50 concerning [,] and [.] as the last character). When a value is moved to such a field, the character positions in the field corresponding to [,] B 0 or [.] in the Picture have the characters [,] space 0 [.] respectively inserted into them. The characters of the value are placed around these insertion characters and are aligned as with unedited NUMERIC fields according to the decimal points of the source and the receiving fields.

THE CURRENCY SYMBOL

Either a single or a succession of currency symbols can appear in a Picture. The symbol used in the Picture is £ or $. The rules for the two are the same except that if $ is used, the symbol $ appears whenever the following rules indicate that the object program produces £.

If the Picture contains one occurrence of £, the corresponding position in the field becomes £ whenever a value is moved to the field.

If the Picture contains a succession of the character £, all but the first represent character positions into which characters of a value are moved. When a value is moved to a field with such a Picture, the contents of the field after the move has taken place are as if the following operations had occurred.

1 The value is moved into the field and insertion symbols are inserted.

2 All leading non-significant zeroes to the left of the assumed position of the decimal point, whose corresponding Picture symbol is £, are converted into spaces. A space is put into the position represented by the leading £ in the picture.

3 Every insertion symbol to the left of the assumed position of the decimal point which has not a numeric digit to its left is replaced by a space.

4 The rightmost space placed in the field by operations 1 and 2 above is replaced by the symbol £.

5 If the field has usage DISPLAY-1 or DISPLAY-4 and the tens of shillings position contains zero, the zero is replaced by a space. If the field has usage DISPLAY-2 or DISPLAY-5, the tens of shillings and tens of pence positions are both examined. If either contains zero, the zero is replaced by a space.

EXAMPLES

A field has a Picture of £999.V9 and usage DISPLAY:

If the value 123.4 is moved to it, it will contain £123.4.

A field has a Picture £££99 and usage DISPLAY:

If the value 12345 is moved to it, it will contain £12345.

If the value 00345 is moved to it, it will contain ▽▽£345.

If the value 00005 is moved to it, it will contain ▽▽▽£05.

If a Picture contains £ to the right of the assumed decimal point position, see also page 126.

REPORT SIGNS

The report signs CR DB [+] and [-] can appear in a Picture. Only one of the four possibilities can appear in one Picture. However, a succession of the character [+] or a succession of the character [-] is possible.

The symbol CR represents two character positions in the field which contain spaces if the value of the field is greater than or equal to zero; otherwise, the character positions contain CR. For example, a field has a Picture 999CR

If the value 123 is moved to it, it will contain 123▽▽.

If the value 000 is moved to it, it will contain 000▽▽.

If the value -123 is moved to it, it will contain 123CR.

The symbol DB has the same meaning as CR except that the characters DB appear in negative fields instead of CR.

If a single [+] appears in a Picture, it represents a character position which will contain [+] if the value moved to the field is greater than or equal to zero and [-] if the value moved to the field is less than zero. A single [+] can appear in a Picture to the left of a succession of currency symbols. If the currency symbol is floated to the right in the field as described above, the position corresponding to the [+] in the Picture is unaffected and records [+] or [-] as described above. A single [+] can appear in a Picture to the left of Z or [*] (see page 126). For example, a field has a Picture +£££99. If the value 123 is moved to it, it will contain +▽▽£123.

If the Picture contains a succession of the character [+] all but the first represent character positions into which characters of a value are moved. When a value is moved to a field with such a Picture the contents of the field after the move has taken place are as if the following operations had occurred:

1 The value is moved into the field and insertion symbols are inserted.

2 All leading non-significant zeros to the left of the assumed position of the decimal point whose corresponding Picture symbol is [+] are replaced by spaces. The leading sign is replaced by a space.

3 Every insertion symbol to the left of the assumed position of the decimal point which has not a numeric digit to its left is replaced by a space.

4 The rightmost space placed in the field by operation 1 or 2 is replaced by the sign [+] or [-] according as the value moved to the field is not less than or less than zero respectively.

5 If the field has usage DISPLAY-1 or DISPLAY-4 and the tens of shillings position contains zero, the zero is replaced by a space.

6 If the field has usage DISPLAY-2 or DISPLAY-5, the tens of shillings and the tens of pence positions are both examined. If either contains zero, the zero is replaced by a space.

EXAMPLES

A field has a Picture +999.V9 and usage DISPLAY:

If the value 123 is moved to it, it will contain +123.0.

If the value 000 is moved to it, it will contain +000.0.

If the value -123 is moved to it, it will contain -123.0.

A field has a Picture ++,+++V.9 and usage DISPLAY:

If the value 123 is moved to it, it will contain ▽+123.0.

If the value 003 is moved to it, it will contain ▽▽▽+3.0.

If the value -0.6 is moved to it, it will contain ▽▽▽▽-.6

The character [-] in a Picture has the same meaning as the character [+] except that where [+] would cause the corresponding field to contain [+] the character [-] would cause it to contain space, that is to say, a negative value would be indicated by [-] and a positive value or zero by space.

If a Picture contains a succession of the symbol [+] or [-] and they extend to the right of the assumed decimal point position see also page 126.

ZERO SUPPRESSION

The symbol Z can appear in a Picture to represent leading positions to be occupied by the characters of a value moved to the corresponding field. The same meaning is conveyed by the phrase ZERO SUPPRESS..... (see page 45).

If zero suppression is specified for a field by either of these means, the contents of the field after a value has been moved to it is given by considering that the following operations have occurred:

1 The value is moved into the field and insertion symbols, if any are specified, are inserted. The Picture character Z is taken to have the same meaning as 9 for this purpose.

2 All leading non-significant zeros to the left of the assumed position of the decimal point, whose corresponding Picture symbol is Z, are replaced by a space.

3 Every insertion symbol which has not a numeric digit to its left is replaced by a space unless it is in the position of the assumed decimal point.

4 If the field has usage DISPLAY-1 or DISPLAY-4 and the tens of shillings position contains zero, the zero is replaced by a space.

5 If the field has usage DISPLAY-2 or DISPLAY-5, the tens of shillings and tens of pence postions are both examined. If either contains zero, the zero is replaced by a space.

EXAMPLES

A field has a Picture ZZ,ZZ9.V99 and usage DISPLAY:

If the value 01234.56 is moved to it, it will contain ▽1,234.56.

If the value 00234.56 is moved to it, it will contain ▽▽▽234.56.

If the value 00004.56 is moved to it, it will contain ▽▽▽▽▽4.56.

If the value 00000.56 is moved to it, it will contain ▽▽▽▽▽0.56.

If the value 0.56 is moved to it, it will contain ▽▽▽▽▽0.56.

If the character Z occurs to the right of the character V in the Picture (i.e. to the right of the assumed position of the decimal point) see also below.

A single [+] or [-] can appear in a Picture to the left of a Z. If leading zeros are replaced by spaces in the corresponding field, the position corresponding to the [+] or [-] is not affected and becomes [+] or [-] or space according to the rules for report signs (see page 124).

CHEQUE PROTECTION

The character [*] in a Picture or the clause $\left\{\begin{matrix}\text{CHECK}\\\text{CHEQUE}\end{matrix}\right\}$ PROTECT has exactly the same meaning as the character Z in a Picture or the clause ZERO SUPPRESS (see page 45) with the following exceptions:

1 Whenever Z or ZERO SUPPRESS causes a zero to be replaced by a space it is replaced instead by [*].

2 The clause BLANK WHEN ZERO is ignored by the compiler if it is used to describe a field whose Picture contains [*] or which is described by $\left\{\begin{matrix}\text{CHECK}\\\text{CHEQUE}\end{matrix}\right\}$ PROTECT.

3 If the character [*] appears in a legal Picture to the right of the character V, (that is, to the right of the assumed position of the decimal point) and the value zero is moved to the corresponding field, all the character positions of the field are set to the value [*].

EXAMPLES

A field has a Picture *,***.V*;

If the value 0123.4 is moved to it, it will contain **123.4.

If the value 0003.4 is moved to it, it will contain ****3.4.

If the value 0000 is moved to it, it will contain *******.

BLANK WHEN ZERO

The meaning of this clause is also implied by a Picture which contains Z or + or - or £ in every character position to the right of the character V.

If a field which is described as BLANK WHEN ZERO has the value zero moved to it, it contains a space in every position of the field after the move has taken place, overriding all other editing requirements of the field (however, see Cheque Protection above). If the character Z appears in the Picture to the right of the assumed decimal point and the value zero is moved to the field the result is identical with that which would have been obtained, had the field been described as BLANK WHEN ZERO.

Chapter 11 Representation of data on external media

CHARACTER PERIPHERAL DEVICES

Records held on character peripheral devices are normally declared in character form, that is, none of the fields of a record have usage COMPUTATIONAL or COMPUTATIONAL-1. If fields are declared as COMPUTATIONAL or COMPUTATIONAL-1, no automatic conversion from character to binary form is implied when the information is read, or from binary to character form when the information is punched.

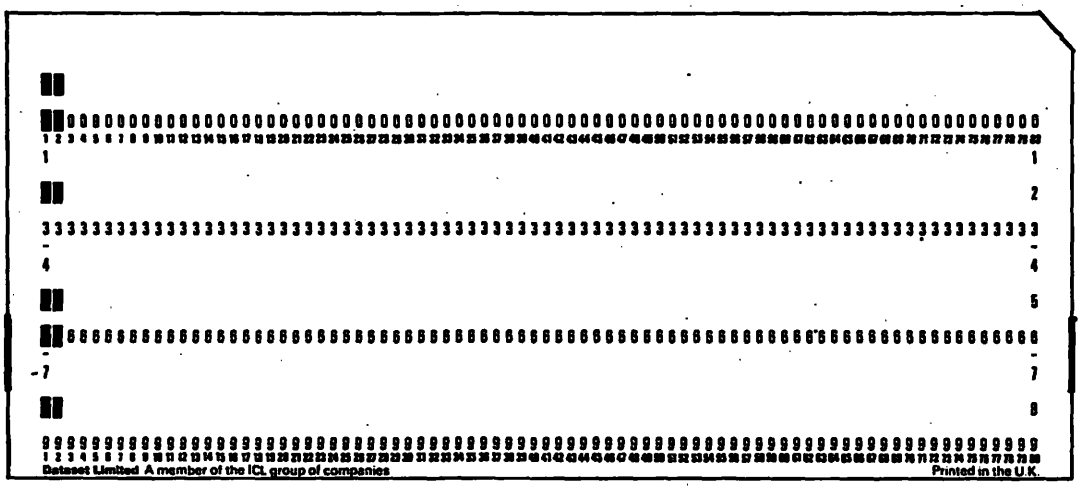
No automatic distribution of information takes place either immediately after information is read or immediately before it is punched. If the programmer wishes, for example, to record variable length fields which exclude non-significant leading zeros, he must treat the record as fields each one character long and program the distribution explicitly. The most convenient way to do this may be to incorporate a subroutine previously translated to semi-compiled form.

Punched cards

The standard way of recording information on punched cards is such that each column of the card represents a single character. When a card is read the information recorded in it is copied into the record area in the computer store. If no special recording mode is requested by the COBOL programmer, the first character position of the record area contains the character punched into the first column of the card, the second character position contains the character punched into the second and so on. The record area assigned in store is as long as the longest record declared for the file. This longest record must not be longer than one card. Each record starts on the first column of a new card.

If non-standard size cards are to be read, the programmer need only assign the appropriate card reader (see page 28) and declare records of the appropriate size.

When using a binary image card reader the End Card should be punched thus in columns 1 and 2 (equivalent to four asterisks in columns 1 to 4):



Paper tape

Only one form of recording information on paper tape is recognized by a COBOL program. Information must be recorded in character form and only alpha-shift characters are allowed; that is to say, the character set is the one given in Appendix 1 excluding the characters $ [ ] ↑ ← .

When records are punched on paper tape by means other than the computer, either the full number of characters for the largest record in the file can be punched or less than that number can be punched followed by a newline character. The maximum size of a record is 128 characters including the two characters of the newline character if it is present.

When a record is read from paper tape, the first character read is stored in the first character position of the record area, the second in the second, and so on, until either the record area is full or a newline character is met or both. The newline character if present is stored as two characters. At the start of the reading operation alpha shift is assumed. The programmer must take care to allow *two* characters of storage for a newline character if it is to appear.

When records are punched into paper tape by the computer under the control of a COBOL program, punching continues until as many characters as are in the largest record for the file have been punched. At the start of the punching operation, alpha shift is assumed. A newline character is not punched at the end of the record unless the program places it in the record area explicitly as two characters.

VARIABLE LENGTH FIELDS ON PAPER TAPE

Variable length fields on paper tape may be processed by means of the subroutines described in Chapter 12, In addition, the programmer can effect the distribution of fields by explicit coding. He should define his record as:

```
01    PT-RECORD
02    CHAR PICTURE X OCCURS .... TIMES
```

Then given a field POINTER initially equal to 1, he can examine the characters in turn using CHAR (POINTER) and updating POINTER. In this way he can detect field-terminators and newline symbols.

Output files can be processed in a similar manner. If the programmer requires anything other than an exact copy of the record as it is in store to be punched in paper tape, he must place the record in the form required in the record area of the output file before the WRITE statement is executed. For example, using the method described above, the programmer can make up variable length records and at the end insert a newline symbol followed perhaps by runout symbols.

Printing

Each internal character is printed as a separate character and must belong to the character set given in Appendix 1. A record occupies one complete line on the printer starting at the leftmost printing position. The programmer can select the kind of printer to be used by writing an appropriate SELECT . . .ASSIGN clause (see page 27). If the record is longer than the print line, the compiler will print a warning and information will be lost from the right-hand end. If the print line is longer than any record in the file, unpredictable characters may be printed to the right of the record.

The programmer must use the ADVANCING option to cause the printer to space between records. If he does not do so, successive records will be printed on the same line.

MAGNETIC TAPE

Information is recorded on magnetic tape according to the conventions of the 1900 Series Storage Device Housekeeping system. Each file is preceded by a standard Header Label, the contents of which the programmer can control to some extent (see page 37). Information is held in blocks of one or more records, the size of which is controlled by the programmer (see page 37). The first word of every record must be a binary field holding a value which represents the size (in words) of the record.

Variable length records

See *Variable length records in magnetic files* on page 130.

DIRECT ACCESS DEVICES

1 The 1900 COBOL direct access sytem uses the Direct Access Housekeeping package at the logical level. This means that the COBOL user must adhere to the information standards and formats described in Chapter 16, (page 103) of *Direct Access.* It also means that both the basic and extended software facilities are available to him.

2 The user can process in COBOL programs serial, sequential and random files, either by using the ENTERed subroutines described on pages 149 to 154.1, or by using the COBOL input-output verbs and the FILE-CONTROL options described on pages 29 to 31. (The terms serial, sequential and random are to be understood in the sense defined for them in the Direct Access manual). All direct access files processed in COBOL programs, however, must be permanent files, that is, they must have first been allocated by means

of the File Allocator program. Details of this program may be found in the 1900 Series manuals *Direct Access* and *Direct Access Utilities.*

No distinction is made in the COBOL source program between serial and sequential files. However, the distinction must be specified when allocating the file. When using FILE-CONTROL options ACCESS MODE IS SEQUENTIAL is written in the COBOL program for both types of file. The effect of the input/output statements depends upon the way in which the file is stored. Hence "next record" for a file stored serially means the record in the next physical position on the device, whilst "next record" for a file stored sequentially means the next record in key-sequence order.

Sequential files with standard index tables can also be processed in COBOL programs. These files can be accessed in three ways, sequentially, randomly or both methods of accessing can be combined in the same program. When using FILE-CONTROL options if it is desired to access such a file only sequentially and no records are to be inserted, ACCESS MODE IS SEQUENTIAL is written and the file can then be treated as a sequentially stored file without index tables. If insertions are to be made in such a file ACCESS MODE IS RANDOM and ORGANIZATION IS INDEXED must be specified. If it is required to access such a file randomly the clauses ACCESS MODE IS RANDOM, ORGANIZATION IS INDEXED and SYMBOLIC KEY must be written and the key of the specific record required must be set in the SYMBOLIC KEY field and a SEEK statement executed before each access of the file. If, after accessing a specific record (or immediately on opening the file), it is required to access the next record of an INDEXED file, it can be done by executing a READ statement not preceded by a SEEK statement. In this way a series of records in key-sequence order starting at a specific record can be accessed.

3 As with magnetic tape files, a field with PICTURE 1(24) must be described at the beginning of every record of a direct access file unless the RECORDING MODE V or F option is used. This is used by the direct access system to hold the size of the record. In the case of direct access records, however, the twelve most significant bits of the field are used to hold other information so that if the user wants to ascertain the size of an input record from this field, he must describe it as having two subfields, each with PICTURE 1(12). In this way he can access the twelve least significant bits.

4 A number of exception conditions from which it is possible to recover can arise in processing direct access files. Some of these are recognized automatically, e.g. the end-of-file and invalid key conditions, but others must be tested for explicitly in the source program. This is done by associating a data-name with the file in an APPLY .... TO TRANSFER-REPLY clause and testing this field after an input/output statement has been executed. The exception conditions and the corresponding values of the reply field are given on page 31. Since the possible values of the reply field are 0 to 6 and 0 corresponds to the case where no exception has arisen, the field may be tested by means of a GO DEPENDING statement. This APPLY clause should be used only in associated with COBOL verbs and FILE-CONTROL options, not when using ENTERed subroutines.

5 Overflow may be handled automatically by the object program by one of the following methods

1 by means of the FILE-CONTROL options FIRST-LEVEL and SECOND-LEVEL (see page 30)

2 by means of the subroutines CLEVEL1 and CLEVEL2 (see page 150-151)

In the first case the user may specify one of the following:

no automatic overflow handling

first-level handling only

second-level handling only

first and second level handling

In the second case, however (when using the subroutines) the user cannot specify second-level handling only.

When an overflow occurs that is not to be handled by the object program then this is indicated in the reply word.

The user can then,

(a) open a spare file, if using COBOL verbs and FILE-CONTROL options

or

(b) if using subroutines, either open a spare file or extend the existing file, using CEXTEND (see page 154).

If a spare file is opened, overflow records can be written to it and the two files later merged by a run of the file re-organizer utility program. When using subroutines the spare file may be held on the device with zero length and extended when necessary by means of CEXTEND. It can be contracted back to zero length by the file re-organizer program after the two files have been merged.

Extension is not available by means of FILE-CONTROL options. Overflow need not be specified for files used as input only, since Direct Access Housekeeping Reads follow tags to overflow buckets.

6 A number of error-conditions from which it is not possible to recover are detected at run-time by the Direct Access Housekeeping package and messages corresponding to them are displayed on the console typewriter. The messages and the conditions in which they arise are given in the *Direct Access* manual. The file number in these messages is the same as the integer specified after the hardware name in the SELECT . . . . ASSIGN statement for the file.

7 See Appendix 4 for a table summarizing the clauses and statements necessary for processing direct access files by means of the 1900 COBOL direct access system.

VARIABLE LENGTH RECORDS IN MAGNETIC FILES

The processing of variable length records on magnetic tape or disc can be achieved by various methods, including the use of the OCCURS . . . DEPENDING clause.

To conform to 1900 Series practice, the first word of every record held on magnetic tape or disc must contain the size of the records in words. The RECORDING MODE clause in an FD description obviates the need for a programmer to define this field at the beginning of each record (see page 36).

If the RECORDING MODE clause is not used, then the programmer is responsible for defining the count-word at the beginning of each record of the file, e.g.

02 data-name PICTURE 1(24) COMPUTATIONAL-1.

where the name of the field is devised by the programmer and may appear in Procedure Division statements.

Where the RECORDING MODE clause is not used the programmer can control the number of words written by inserting the record size in the first word of the record. The coding generated by the COBOL WRITE, REWRITE and RELEASE verbs examines the first word of each record to be written to magnetic tape. If the word contains zero, the coding moves the number of words in the record to it. If it contains other than zero, it does not change it. In either case, the word is zeroized immediately the record has been written. Care must be taken not to move a value to the first word, for example in a group move or when moving spaces to clear the record area.

If the OCCURS . . . DEPENDING clause is written the programmer is responsible for the contents of the dependent field but need not update the contents of the count-word.

This facility provided by OCCURS . . . DEPENDING is limited in that there can only be one variably occurring field in a record and it must be the last elementary field in the record or a group field containing the last elementary field in the record (see page 48). Methods of achieving variable length working without using the OCCURS . . . DEPENDING clause, are described below.

If the OCCURS . . . DEPENDING option is not available the following method may be employed.

Tables of fixed length fields may be specified by using the option.

OCCURS integer TIMES

Such a statement may be made about any field of a record but, if it is made about the last field of a record, the programmer can obtain most of the advantages of variable length tables. If he moves a value to the first word of the record before it is written, the programmer can control the amount of information to be written to magnetic tape. If this is done, the value must be referred to when the record is being read in order to determine how much of the record and therefore of the table has been read.

The above allows the programmer to have a table with a varying number of items at the end of a record. Such tables elsewhere in records generally produce inefficient programs. It may, however, be necessary to have two or more tables at the end of a record which, together, will never occupy more than a particular number of words even though the sum of all their maxima is greater than that number. Provided that the items in all the tables are the same size, an arrangement such as the following can be used.

```
02  SIZE-1 .....
02  SIZE-2 .....
02  SIZE-3 .....
02  ITEM-1 ....OCCURS ....TIMES
02  ITEM-2     REDEFINES   ITEM-1 .....OCCURS ....TIMES
02  ITEM-3     REDEFINES   ITEM-2 .....OCCURS ....TIMES
```

In this record description SIZE-1 contains the number of occurrences of ITEM-1, SIZE-2 those of ITEM-2 and SIZE-3 those of ITEM-3. It is assumed that ITEM-2 follows ITEM-1 and that ITEM-3 follows ITEM-2 in all the records. A subscript for ITEM-2 could then be calculated by adding the value of SIZE-1 to the particular subscript number. Similarly a subscript for ITEM-3 could be calculated from SIZE-1 and SIZE-2.

It would also be possible for the elements of subsequent tables to be smaller than those that precede them provided they were exact sub-multiples.

True variable record handling (i.e. more than one variable length field in a record or a variable length field in the middle of a record) can be achieved by using the subroutines described in Chapter 13.

Where different types of records are to be written, each type being fixed length, but of a different size to the other types the programmer need not make any special provision for specifying the size. By using different record descriptions for each type of record the size inserted in the record will be that calculated from the record descriptions in the DATA DIVISION.

Chapter 12 Programming techniques

FORMAT DISCIPLINES

Although COBOL is written in an essentially 'free form' (subject to the restrictions given in Chapter 15), the user will nevertheless reap many advantages from a few self-imposed format disciplines. It is suggested that these should include the following:

1 All field names in the Data Division should be indented one or two columns beyond the immediately preceding field name, if the latter is at a higher level, e.g.

```
01  AAA....
 02  BBB....
  03  CCC....
  03  DDD....
 02  EEE....
```

This enables the logical structure of the record to be easily understood from the printout.

2 The use of continuation lines, i.e. the splitting of words across two lines is not recommended unless there are special circumstances.

3 If the user assigns his own sequence numbers, they should be in steps of 10 or 100 to facilitate the insertion of correction lines without disturbing the original sequencing.

4 Each sentence should begin on a new line. Compare the following two examples:

(A) CALCULATE-PROCUREMENT. IF REVIEW-DATE GREATER THAN TODAYS-DATE GO TO NEW-MAST-OUT. ADD FORECAST TO RESERVE. SUBTRACT STOCK-ON-HAND FROM NEW-AMOUNT. MOVE NET-AMOUNT TO NET-ORDER. (etc.)

(B)
```
CALCULATE-PROCUREMENT.
    IF REVIEW-DATE GREATER THAN TODAYS-DATE GO TO
    NEW-MAST-OUT.
    ADD FORECAST TO RESERVE.
    SUBTRACT STOCK-ON-HAND FROM NET-AMOUNT.
    MOVE NET-AMOUNT TO NET-ORDER. (etc.)
```

Among the advantages gained by writing the source program in form (B) rather than (A) are:

(a) It is easier for the programmer or someone reviewing the program to follow the logic.

(b) Corrections are simplified. For example, if FORECAST has to be subscripted, only one line has to be corrected.

(c) Statements or sentences can be inserted, deleted or replaced without distrubing the structure of the paragraph.

(d) Statements within the paragraph may be referenced by other sections of the program by assigning a new paragraph name to them, which can easily be inserted as a new line.

THE DATA DIVISION

Data names

Meaningful data names should always be used instead of mnemonics or abbreviations, e.g.

SUB-TOTAL-A instead of SUBTOTA

ACCOUNT instead of ACCT

The user should remember, however, that the longer the data name (or procedure name), the more space it will occupy in the Names table that the compiler creates and uses. In a large program this will increase the chance of exceeding the compiler limit for the table, although it should be noted that the compiler will:

(a) allocate more space to this and other tables, if possible, when the original allocation has been filled, or

(b) allocate space according to parameters supplied by the user for a compilation (*DATA and *PROCEDURE Steering lines – see page 190.1).

In general, meaningful data names can usually be made up of, say, fifteen or less characters and should be so made up.

Magnetic tape files: the record count word

As stated on page 130 unless the RECORDING MODE clause is used, in which case the compiler will insert count-words at the beginning of each record, all files assigned to magnetic tape should have an extra field defined at the beginning of each of their records with the description PICTURE 1(24) to accord with the requirements of the 1900 series Magnetic Tape Housekeeping package. The user need not refer to this area unless he wishes to process certain types of variable record (see page 130) and if he does not refer to it, it can be given the name FILLER.

If the user does not define such an area, or if it is mis-defined or its contents mutilated at run-time, the object program will stop with an error message printed by the Magnetic Tape Housekeeping package: HALTED F. This error will most frequently result from MOVing into a record area (01 level), thus overwriting the contents of the word-count field in the 02 level.

Magnetic tape files: the block size

If the BLOCK n CHARACTERS clause is used in a File Description (FD level) a block area will be allocated in the object program equivalent in size to the number of characters specified rounded up to the next word.

If the BLOCK n RECORDS clause is used, a block area will be allocated equivalent in size to 'n' times the largest record in the file. It does *not* ensure that a block as written or when read will contain 'n' records. If there are smaller records defined in the file then the block may contain *more* than 'n' records. This is of no consequence to the user except to note that a COBOL object program, when recreating a file originally produced by a non-COBOL program, will not reproduce the large variations in block size which the file may originally have shown. It will instead produce variable length blocks containing a variable number of records, all the blocks tending to be near the maximum size possible.

No limit other than the store size of the object computer is imposed on the size of the blocks of magnetic tape output files, either by the COBOL compiler or the Magnetic Tape Housekeeping package which it employs. However if such files are to be processed by utility routines, e.g. Sorting and Listing, the block size should not be greater than 512 words.

Paper tape: end marker

The End Marker (****newline) for data blocks on paper tape input to COBOL object programs must always be followed by another newline character, or alternatively there can be two End Markers (****newline****newline). This is necessitated by double buffering on the object program.

Paper tape: inter-block gaps

The coding generated for WRITE statements for paper tape files will normally create data blocks *without* gaps between them. If the user wishes to create blocks *with* gaps between them, he should:

(a) define and name an extra DISPLAY field at the end of his output record, of a multiple of two characters in size (two or four or six etc.),

(b) MOVE into this field a literal consisting of the pair of characters ↑▽ repeated as many times as he wishes there to be run-out characters in the gap, e.g. for a gap of three characters,

MOVE '↑▽↑▽↑▽' TO FIELD

where ▽ is space character and where the field is defined as six characters in length.

Short data blocks on paper tape or cards

If a data input file consists of blocks of less than four characters (as specified in the BLOCK clause of the FD level, and the descriptions of the record), the normal object code generated by the AT END clauses of READ statements addressing the file will not succeed in detecting the standard End Marker (****). If the user does not wish to increase the block size in the file description to four characters, he must therefore write his own end-of-file test in the source program after each READ statement addressing the file. He can test the block just read against a literal of three asterisks, or indeed any other three-character literal which he may choose as his End Marker, and which must of course be inserted at the end of the input file when it is created. This test should be named as a paragraph, and the AT END branch of the READ should GO to this paragraph.

Fillers

It is fairly important for object code efficiency that fields should not be described as being longer than necessary. For example, it would appear that if the first ten columns on a card contained a number of some kind, it would be described as

```
01  CARD-RECORD.
02  NUMBER PICTURE 9(10).
```

However if it were known that the number could not exceed 999999, it would be better to describe it as

```
01  CARD-RECORD.
02  FILLER PICTURE X(4).
02  NUMBER PICTURE 9(6).
    (etc.)
```

This would save code if arithmetic operations were to be performed on NUMBER, since the converted value would be single, not double length, and the code generated to access and convert it would be considerably less. If the user is not short of space, this does not matter too much; but it is as well to remember that in general for this and other similar situations fields should not be described as greater than necessary.

For other remarks on FILLERs in SYNCHRONIZED and bit fields, see below.

USER'S DATA ALLOCATION

The following remarks draw attention to the optimal use of certain COBOL functions.

COMPUTATIONAL fields

To perform arithmetic on fields represented on character form, e.g. DISPLAY NUMERIC, the compiler-generated object coding converts the contents to a form that meets the requirements of the 1900 series. This form is the same as that in which COMPUTATIONAL fields are held. If a DISPLAY NUMERIC field is referred to in several arithmetic statements, the contents must each time be unpacked and converted. It will therefore produce much faster object coding if the user defines a WORKING-STORAGE field as COMPUTATIONAL with exactly the same picture as the DISPLAY NUMERIC, MOVEs the DISPLAY field to it and makes all the arithmetic statements address it instead. The effect is even better if the field is described as SYNCHRONIZED RIGHT as well as COMPUTATIONAL, since packing and unpacking may be simpler and the field is in fact stored in the form most convenient for fast arithmetic on the 1900 series (but see below for qualifications on the use of SYNCHRONIZED).

Although normally, as above, only Working Storage fields will be described as COMPUTATIONAL SYNCHRONIZED RIGHT, it may also be useful to describe fields in magnetic tape output records in this way, when they are subsequently to be read by a COBOL object program. In the latter case the corresponding fields in the input record will be described in exactly the same way.

It should be noted that while fields described as COMPUTATIONAL-1 are held in true binary form, arithmetic performed on these fields is on the whole slower than arithmetic performed on binary decimal (COMPUTATIONAL) fields. Furthermore, since the usage COMPUTATIONAL-1 is extremely

machine orientated, it is not recommended unless highly desirable or absolutely necessary (e.g. for processing an input record produced by a PLAN program).

The use of FILLERS in bit fields

Since character (DISPLAY) fields start on character boundaries whereas bit fields begin immediately after the last field and so may start or end in the middle of a character, the user should make sure to define FILLER fields for all those which will be allocated by the compiler though not referenced in the source program. This is particularly important in the case of tables with mixed character and bit fields where failure to insert necessary FILLERs may invalidate the COBOL statements intended to access the tables.

Examples:

(a)
```
02  FRED PICTURE 9 DISPLAY.
02  JOHN PICTURE 1(8) COMP-1.
02  HARRY PICTURE 9 DISPLAY.
```

If FRED is assigned to character C_0, JOHN will occupy C_1 and the first two bits of C_2, and HARRY will occupy C_3. The user should thus have inserted, between JOHN and HARRY, an 02 level field defined FILLER with usages COMP-1 and PICTURE 1(4) in order to define the four bits between the end of JOHN and the beginning of HARRY. If this FILLER is omitted, as in the example, the compiler prints error message 136 (NEEDS FILLER) but nevertheless allocates storage as if it were present.

(b)
```
02  JIM OCCURS 10 TIMES.
03  BEN PICTURE 9.
03  TOM PICTURE 1.
```

Suppose the field before JIM ends on a character, so BEN (1) may correctly fit in the next character. TOM (1) will fit in the first bit of the character after that. However BEN (2) may not immediately follow it but will have to fit in the next character. Thus, from the above description of the table it would appear to be 70 bits long, but it is in fact 120 bits long. The description of TOM should be followed by the binary field '03 FILLER PICTURE 1 (5)'.

As in (a) the compiler will nevertheless make the correct assumptions though flagging an error.

Synchronization

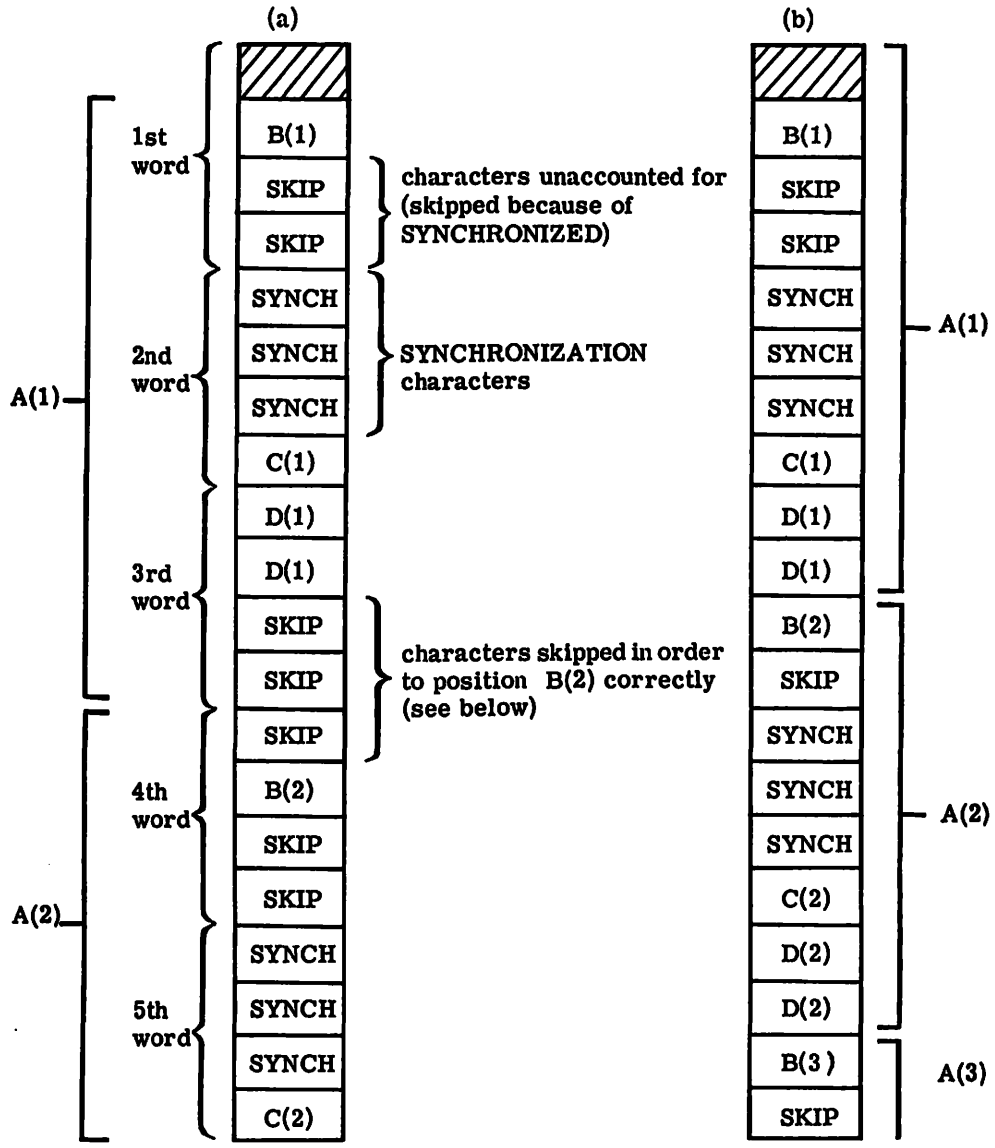
While the SYNC RIGHT option is often useful in producing faster object code (as described above), it too is machine dependent, since the size of a record containing SYNC fields depends on the word size of the object machine. Programs written for other machines containing SYNC fields should therefore be critically examined before compiling them for the 1900 series.

Furthermore, when used in 1900 COBOL programs in defining tables areas there are additional possibilities of error unless the rules are fully understood.

Suppose a table is defined as:

```
02  A  OCCURS 10 TIMES.
03  B  PICTURE 9.
03  C  PICTURE 9 SYNC RIGHT.
03  D  PICTURE 99.
```

Suppose too that the field before the table ends on the first character within a word, then the compiler will assume the table will look like diagram (a) on the opposite page.



This example illustrates two points. Firstly, all storage should be accounted for explicitly. The user should have inserted a two-character FILLER between B and C. Secondly, note how B(2) is assigned to the second character in the fourth word; the user should have inserted another FILLER of three characters after D. The compiler has assigned storage correctly despite the absence of these FILLERS, though flagging errors, but it will probably escape the user's notice that his table occupies not 70 characters but 119.

It is important to notice that if the compiler had not inserted the three SKIP characters after D, the layout and size of A(1) would have been different from A(2), as in (b), and the table would not have been correctly processed.

DISTRIBUTION OF INPUT DATA

The 1900 COBOL READ verb does *not* generate code to distribute input data according to record type. It is unnecessary and undesirable for the compiler to do this because the COBOL programmer is thus left free to decide for himself how to identify the different types of record where a file contains more than one type. For example, there might be two types of record, defined thus:

```
FD  A.
01  A1.
02  TYPE PICTURE X.
02  A11      PICTURE XX.
02  A12      PICTURE 999.
01  A2
02  FILLER  PICTURE X.
02  A21      PICTURE 9.
02  A22      PICTURE 9999 DISPLAY-1.
02  A23      PICTURE S9999.
```

These two records are of different size and structure. Note, however, that there is a field in each record which occurs in the same relative position from the start of the record and is of the same size, i.e. TYPE and FILLER. (In this case they both come at the start of their respective records.) Since records in the File Section automatically redefine one another, on 'reading next record', either the field TYPE or the field FILLER will fall in the first character of the record area. Suppose records of types A1 and A2 are identified by having their first character equal to Q and P respectively. The programmer might then write, for example:

```
        READ A
        IF TYPE = "P" GO TO ITS-P.
        DISPLAY A11, A12.
        STOP RUN.
ITS-P.
        ADD A21 , A22 GIVING A23.
        DISPLAY A23.
        STOP RUN.
```

Thus having made the test, the program jumps to a set of statements which address the record area in terms of the appropriate description or mapping.

Of course, if one were to REDEFINE fields within a record, the same argument against distribution would apply since if it distributed according to the first description, the REDEFINE description would be nonsense. For example,

```
02  A
03  A1   PICTURE 999.
03  A2   PICTURE 999.
02  B    REDEFINES A PICTURE 9(6).
```

Thus it would be possible either to refer to all six digits as one elementary field B or to refer to the first or second half separately as A1 or A2. If, on reading, A1 and A2 were distributed into separate non-contiguous areas, it would make no sense to address B.

DATA VALIDATION

At object run time, if a non-numeric character is read into a numeric field, the character will be ignored (on conversion by MOVEs etc.) and no error is flagged. Because of this the user is recommend to validate his data by using the IF ..... NUMERIC test.

REDEFINES

The REDEFINES clause is useful not only for saving storage by commoning up the space used by two fields which are needed at different times during the running of the program, but also as a 'remapping

function' which can achieve what in many cases no combination of verbs could achieve. Consider the following examples:

1 VALIDATING DATA Suppose given a 50-character alphanumeric field, FRED, it is required to check that the thirteenth character is a Q.

```
02   FRED     PICTURE X(50).
02   JIM      REDEFINES FRED; PICTURE X; OCCURS 50 TIMES.
IF JIM (13)=  "Q".....
```

or alternatively:

```
02   FRED     PICTURE X(50).
02   ALEX     REDEFINES FRED.
03   FILLER   PICTURE X(12).
03   BILL     PICTURE X.
03   FILLER   PICTURE X(37).
IF BILL =     "Q".....
```

Suppose it was necessary to check that all the characters of FRED were in the range A to H. Redefining FRED as JIM above, also define

```
77   PROX     PICTURE 99 COMPUTATIONAL
              SYNCHRONIZED RIGHT
              VALUE IS 1.
START.
    IF JIM (PROX) LESS THAN "A" GO TO BAD-CHAR.
    IF JIM (PROX) NOT GREATER THAN "H" GO TO OK.
BAD-CHAR.
    DISPLAY "CHARACTER" PROX "OF FRED =" JIM (PROX).
OK.
    IF PROX NOT EQUAL TO 50 ADD 1 TO PROX
    GO TO START.
      (etc.)
```

So if the sixth character of FRED is K, it will print

```
CHARACTER 6 OF FRED = K.
```

Thus whilst taking no extra store, the remapping function achieves what no combination of verbs could achieve.

2 TABLES OF VALUES Suppose it is required to set up in Working Storage a table of six COSTS with values 6, 7, 12, 3, 15, 9. It could be done as follows:

```
02   COST-VALUES.
03   C1 PICTURE 99 VALUE IS 6.
03   C2 PICTURE 99 VALUE IS 7.
03   C3 PICTURE 99 VALUE IS 12.
03   C4 PICTURE 99 VALUE IS 3.
03   C5 PICTURE 99 VALUE IS 15.
03   C6 PICTURE 99 VALUE IS 9.
02   COSTS REDEFINES COST-VALUES PICTURE 99 OCCURS 6 TIMES.
```

Thus COSTS (5) equals 15.

3 NON-STANDARD EDITING Suppose it is required to print a sterling result with, say, slashes between the pounds and shillings and the shillings and pence. Define the result with spaces in place of the slashes as follows:

```
02  RESULT          PICTURE 999B99B99   DISPLAY-2.
02  SLASH-RESULT    REDEFINES RESULT.
03  FILLER          PICTURE 999.
03  SLASH1          PICTURE X.
03  FILLER          PICTURE 99.
03  SLASH2          PICTURE X.
03  FILLER          PICTURE 99.
```

MOVE source TO RESULT or ADD sources GIVING RESULT.

MOVE "/" TO SLASH1.

MOVE "/" TO SLASH2.

4 NON-STANDARD SIGNING Some card users, particularly on the Continent, represent the sign in signed fields by an overpunch over the last character of the value of the field on the card. Given such a field A, four characters long, suppose it is desired to convert it to DISPLAY-3 form. Define A as:

```
02  A.
03  A1  PICTURE 999.
03  A2  PICTURE S9.
```

Define a field B in working storage as:

```
01  B       PICTURE S9999 DISPLAY-3.
01  C       REDEFINES B.
02  FILLER  PICTURE X.
02  C1      PICTURE 999.
02  FILLER  PICTURE X.
```

MOVE A2 TO B.

MOVE A1 TO C1.

B now contains the value in DISPLAY-3 form. If it was wanted in DISPLAY NUMERIC form, redefine C as:

```
01  D       REDEFINES C.
02  FILLER  PICTURE X.
02  D1      PICTURE S9999.
```

And follow the two MOVEs by:

MOVE B TO D1.

Now consider the converse problem. Suppose there is a DISPLAY NUMERIC or DISPLAY-3 field, A, with PICTURE S9999. Define:

```
01  X.
02  X1  PICTURE 999P.
02  X2  PICTURE S9.
```

MOVE A TO X1.

MOVE A TO X2.

This involves no REDEFINES but is given for completeness.

Note: It should not be forgotten that records in the File Section automatically redefine each other within the same file.

Possible misuse of REDEFINES

On a 1900 Series machine when a binary number is held in two words, the most significant bit of the second word is a 'null' bit (i.e. it is always zero) and is never taken into account when arithmetic operations are performed upon such a number.

This fact is reflected in 1900 COBOL in the fact that in COMPUTATIONAL-1 fields the 24th bit from the right-hand end of the field is a 'null' bit and is never considered when arithmetic operations are performed upon it. However, if such a field is non-integral and is then REDEFINED, such that the integral and non-integral parts become separated and then arithmetic operations are attempted upon the integral part, the results are in error because the 'null bit' is no longer where it should be (i.e. the 24th bit from the right-hand end of the field). Even the simplest of MOVES in such a case will give an erroreous result.

EXAMPLE

```
02  A   PICTURE 1(40)V1(4).
02  B   REDEFINES A.
03  C   PICTURE 1 (40).
03  D   PICTURE 1 (4).
```

In A the 'null' bit is the 24th bit from the right-hand end i.e. the *20th* bit of the integral part. If it is attempted to carry out arithmetic operations on C these will give erroneous results because the 'null' bit is in the *20th* bit of the field and not in the 24th bit as expected.

Note:

PICTURE 1(24) has a 'null' bit

PICTURE S1(23) has *not* a 'null' bit

(since the sign bit can legally be in the 'null' bit position).

THE REPRESENTATION OF THE DECIMAL POINT

In ICL 1900 Series COBOL the full stop is generally used in a PICTURE as an insertion symbol ⊡ and the implied decimal point is symbolised by a V. But in standard COBOL, and in ICL 1900 COBOL by specifying the *STANDARD directive in the steering lines for a program (see page 190), all periods in PICTUREs do imply the decimal point, as well as being insertion symbols, and there must not be more than one period in a PICTURE.

Where the standard COBOL programmer would write PICTURE 9⊡99 the ICL COBOL programmer would write PICTURE 9 ⊡V99 or 9V⊡99 unless he uses the *STA option.

For instance, the contents of a field described as:

PICTURE 99⊡99

will, after MOVEing in the literal 48.97 be:

(a) when *STANDARD is in force

48.97

(b) when *STANDARD is not in force.

00.48

A further advantage is that if for any reason the user wishes to insert a full stop at the end of a field by placing a full stop at the end of its PICTURE, he can achieve this

(a) by adding a semi-colon or a comma immediately after the full stop at the end of the PICTURE, if PICTURE is not the last clause describing the field, thus:

PICTURE 999⊡ ;

or

(b) if it is the last clause, by writing an additional full stop after the insertion full stop at the end, thus:

PICTURE 999⊡⊡

However, when the DECIMAL-POINT IS COMMA clause is used, the function of the period and comma are exchanged in PICTURE clauses and in numeric literals (see page 24) and the facility offered by *STANDARD is automatically included. Thus it is unnecessary to use both these options although an error will not be flagged if they are. It would be invalid to describe a field as:

PICTURE 99,V99

When the DECIMAL-POINT IS COMMA clause is used, a numeric literal should be written as

e.g. 62,75 instead of 62 [.] 75

If it is MOVEd into a field described as

PICTURE 999,999

it will be printed as 062,75.

If it is MOVEd into a field described as

PICTURE 999 [.] 999

it will be printed as 000·062 i.e. when DECIMAL-POINT IS COMMA is used, a period in a PICTURE is simply an insertion symbol, and does not imply the position of the decimal point, and furthermore there must not be more than one comma in a PICTURE.

THE PROCEDURE DIVISION

Subscripting

If it is required to process a subscripted field, it will save object coding particularly when the subscript is a variable, to extract the item from the table, move it to a working-storage area and process it in that area.

For example, the sentence

```
IF TABLE-ENTRY (6, INPUT-CODE, BRANCH)
EQUAL TO 10 SUBTRACT TABLE-ENTRY
(6, INPUT-CODE, BRANCH) FROM ITEM-PRICE
MOVE TABLE-ENTRY (6, INPUT-CODE, BRANCH)
TO DISCOUNT.
```

will generate object coding to address the table item using three subscripts in three different places.

The example should rather be written as:

```
MOVE TABLE-ENTRY (6, INPUT-CODE, BRANCH)
TO WORK-1.
IF WORK-1 EQUAL TO 10 SUBTRACT WORK-1
FROM ITEM-PRICE MOVE WORK-1 TO DISCOUNT.
```

The working-storage field to which the subscripted item is moved should have the same USAGE as those of the others with which it is involved – COMPUTATIONAL for arithmetic operations, DISPLAY-1 (or 2) for sterling fields, etc. If the operations in which the subscripted items are involved are 'mixed', it may be useful to define more than one working-storage area, each with the usage appropriate to the operation, and move the item to all of them.

Arithmetic, overflow, and oversize operations

It should be noted that if an arithmetic result exceeds the size of the result field and tne user has not included a SIZE ERROR branch in the arithmetic statement, the result will be indeterminate. If in doubt, therefore, always include a SIZE ERROR branch. This is particularly important in the case of variable subscripts.

In the 1900 compiler the limits on the sizes of operands in COBOL arithmetic statements are 12 decimal digits for DISPLAY and COMPUTATIONAL fields, and 48 binary bits (effectively 46) for COMPUTATIONAL-1 fields (see Chapter 9 for the complete rules of arithmetic). However, it is perfectly possible to program arithmetic beyond these limits *in COBOL* without difficulty, and a few examples are given below.

1 DIVIDE Q INTO P GIVING R

where Q is defined as PICTURE 9(4)

P is defined as PICTURE 9(12)

R is defined as PICTURE 9(12)V9(4)

In this situation the compiler would print the message 408 COMPOSITE TOO BIG, because the result field requires a composite of 16 decimal digits.

The required effect may nevertheless be easily achieved by the following data definitions:.

```
01   R PICTURE 9(12)V9(4).
01   FILLER REDEFINES R.
     02   QUOTIENT PICTURE 9(12).
     02   FRACTION PICTURE V9(4).
77 or 01  PRODUCT PICTURE 9(12) COMPUTATIONAL SYNCHRONIZED RIGHT.
77 or 01  REMAIN PICTURE 9(4) COMPUTATIONAL SYNCHRONIZED RIGHT.
```

The Procedural statements will then be:

```
DIVIDE Q INTO P GIVING QUOTIENT.
MULTIPLY QUOTIENT BY Q GIVING PRODUCT.
SUBTRACT PRODUCT FROM P GIVING REMAIN.
DIVIDE Q INTO REMAIN GIVING FRACTION.
```

For maximum object-code efficiency in this solution one should first MOVE Q into a field described as PICTURE 9(4) COMP SYNCHRONIZED RIGHT, QUOTIENT should be a separate field described as 9(12) COMP SYNCHRONIZED RIGHT and its contents should be MOVEd to the first 12 digits of R at the end of the operation.

It may be noted that if and when COBOL allows a REMAINDER facility the first three of the quoted procedural statements might all be replaced by

```
DIVIDE Q INTO P GIVING QUOTIENT REMAINDER REMAIN
```

2 MULTIPLY A BY B GIVING C

where A is PIC 9(10)

B is PIC 9(8)

C is PIC 9(18)

This may be achieved by these data definitions:

```
01  A   PIC   9(10).
01  FILLER   REDEFINES A.
    02  A1   PIC   9(4).
    02  A2   PIC   9(6).
01  B   PIC   9(8).
01  FILLER   REDEFINES B.
    02  B1   PIC   9(2).
    02  B2   PIC   9(6).
01  C   PIC   9(18).
01  C0   REDEFINES C PIC 9(6).
01  C1   REDEFINES C0 PIC 9(12).
77  W   PIC   9(12)   COMP SYNCHRONIZED RIGHT.
```

and these Procedural statements:

```
MULTIPLY A2 BY B2 GIVING C.
MULTIPLY A1 BY B2 GIVING W.
ADD W TO C1.
MULTIPLY A2 BY B1 GIVING W.
ADD W TO C1.
MULTIPLY A1 BY B1 GIVING W.
ADD W TO C0.
```

3 MULTIPLY A BY B GIVING C

where A is PIC 9(12)

B is PIC 9(6)

C is PIC 9(18)

The solution here is simpler:

```
01  A   PIC   9(12).
01  FILLER   REDEFINES A.
    02  A1   PIC   9(6).
    02  A2   PIC   9(6).
01  B   PIC   9(6).
01  C   PIC   9(18).
```

```
01   C0   REDEFINES C PIC 9(12).
77   W    PIC   9(12)   COMP   SYNCHRONIZED   RIGHT.

MULTIPLY A2 BY B GIVING C.
MULTIPLY A1 BY B GIVING W.
ADD W TO C0.
```

In passing, note from these examples the usefulness of two facilities in 1900 COBOL: the abbreviations PIC and COMP in saving writing time, and the REDEFINES option in FILLER fields (see Chapter 5 for all the options that may be used with FILLERs). Note too that although a level 77 has been used to relate the examples to other compilers with which users may be familiar, in fact the 1900 compiler makes no distinction whatsoever between 01 levels and 77 levels in WORKING-STORAGE: both start on a word boundary.

Segmentation

Even when the user knows his source program will have to be segmented to suit the size of the object computer, it is recommended that for the initial compilation no sections should be defined. Provided a *LIST MAP Steering line has been used, however, a Program Map and Data Map listing will be provided and the user will be able to calculate exactly how many lines of object coding will be generated for each of his sentences and paragraphs. He will then be able to decide the best way of segmenting his source and will insert SECTION lines in the appropriate places for the next compilation. No extra compilation run is involved in this because the initial compilation is always, to all intents and purposes, primarily a validation run to flag format and other errors in a source. (Note, however, that since compilation ceases when the object program reaches 32K in size, the user should segment his source in 2, 3 etc. if he thinks it is likely to exceed 32K.)

It will be appreciated that the size of the overlay area in the object program, and correspondingly the size available for Permanent, depends on the size of the largest segment in the program. It is therefore advisable, in most cases, for the main overlay segments to be roughly similar in size, instead of there being one large and a number of much smaller segments (there will of course be exceptions to this

Paths that are infrequently used or used once only in the program, e.g. Housekeeping and End-of-Run routines, should be in separate overlay segments if the program has to be segmented (see also remarks under the CLOSE verb).

It is also good practice to put all error branches in a separate segment (or segments).

The user may here be reminded of the rules for SEGMENT-LIMIT and for the resetting of program switches (ALTERable GO statements) by means of the ALTER verb:

(a) Sections without priorities, or with priorities up to but not including the segment limit, or up to but not including 50 if no segment limit, go into the permanent part of the object program. No program switches are automatically set or unset during the running of the object program, i.e. they have to be set/unset by ALTER statements.

(b) Sections with priorities from and including the segment limit (if any) and up to but not including 50 go into overlay segments. Program switches in these segments are not automatically set or unset at run-time, i.e. they are treated exactly like program switches in Permanent.

(c) Sections with priorities from and including 50 and up to and including 99 go into overlay segments, and ALTERable GO statements in them are automatically reset at run-time, on entry to the segment

There is an important difference between overlays of types (b) and (c), from the point of view of object-time efficiency. When overlays of type (b) are referenced (e.g. by GO or PERFORM statements) they are *not* called in again if they are already present in store. Overlays of type (c) are always called in again when referenced, even if already present in store, and accessing them is obviously much slower than accessing type (b) overlays.

It is often forgotten, when re-segmenting a program (that is, when an initial compilation has shown that a reshuffling or regrouping of the COBOL sections or paragraphs is necessary to get the object program within the limits of the store available in the object computer), that this may change the order in which the procedures will be performed. In these circumstances it is often necessary to insert extra GO statements to preserve the original order in which the program was designed to run, and the user should very carefully scrutinize his source program with this in mind.

Literals

Frequently used program literals, e.g. those used in updating counts, should be defined as constants in the Working Storage section. The compiler does not keep a table of literals but generates a new constant for each literal when it occurs, irrespective of whether it has occurred before.

The advantage of this approach is that where a literal occurs several times in a program, associated with different types of field (DISPLAY-n, COMPUTATIONAL etc.), instead of having to be converted to the required form at run-time, it is generated in this form at compile-time.

Notes on the verbs

ACCEPT FILE KEYS FROM (MNEMONIC-NAME)

This 1900 COBOL option enables the user to vary the generation numbers of magnetic tape files (whether input or output) at object run-time by supplying parameters (on cards or paper tape) which will *automatically* be read in by the object program. It thus eliminates the need for the operator intervention required by the Compact COBOL compilers in implementing the GENERATION-NO option. This can also be effected by using the GENERATION-NO IS data-name clause (see page 36).

The file whose generation number is to be read in must have the WITH GENERATION-NO clause in its FD description; if it has not, the message

GEN-NO NOT REQUIRED FOR FILE nn

will be displayed at run-time on the console.

The parameters for each file whose generation number is to be read in must be on a single card (or paper tape block, terminated by newline), in the following format, punching from column 1:

ff G*ggggggg* P*pppppp*

where ff is the file number in decimal (see page 247)

G is the character G

ggggggg is a six-character decimal number, zero-filled if necessary, being the requisite generation number.

P is the character P

pppppp is a six-character decimal number, zero-filled if necessary, which is used for direct access files only, when opened in OUTPUT mode. It represents the previous generation of the file and specifies the value of the generation number of the file to be opened.

If the file being referred to is on magnetic tape, only G*ggggggg* is written.

If the file is on a direct access device and being opened in output mode and the GENERATION-NO IS clause is not used, both G*ggggggg* and P*pppppp* must be written. P*pppppp* specifies the generation number of the file to be opened and G*ggggggg* specifies the generation number to be given to the file. If the GENERATION-NO IS clause is used, the G*ggggggg* should be omitted and P*pppppp* written in columns 3 to 9 to specify the old generation number. If the new generation number is entered as a parameter it will be ignored.

The file parameter cards (or blocks) must be terminated by the usual End Marker card or block of four asterisks.

At run-time the ACCEPT FILE KEYS statement causes the program to be suspended and the following message to be typed:

LOAD FILE KEYS ON nn

where nn is the peripheral unit. The operator then loads the appropriate peripheral with the parameters and restarts by typing GO.

If a parameter is met for a file for which there is no WITH GENERATION NO clause, the generation number is ignored and the following message is displayed:

GENERATION NO NOT REQUIRED FOR FILE nn

If a second parameter is read for a file the above message is displayed and the generation number is accepted, overriding the previous one. (Note that the program is not suspended.)

If a parameter is read which has an invalid format or a file number out of range the program is suspended and the following message displayed:

INVALID FILE KEY

The parameter can be corrected and re-fed or ignored. The program is restarted by typing GO.

If the ACCEPT FILE KEYS option is not used in the Procedure Division, but a file nevertheless has the GENERATION-NO clause in its FD description, the result will be as in the Compact compilers, viz there will be a halt when the file is OPENed, the message 'SET FILE nn GENERATION-NO IN WORD 0' will be displayed and the operator will have to take the specified action, using a parameter supplied by the programmer on his Job Sheet.

Note: The files in question must not of course be OPENed before the generation numbers are supplied in an ACCEPT FILE KEYS statement.

In later versions of the compiler the ACCEPT FILE KEYS option will be extended to include *relabelling* of files, when in addition to the generation number the file name and its reel number will be supplied as parameters. The SELECT ... ASSIGN n TAPES option in the Environment Division will then have an added significance since it will specify the number of reels of a multi-reel file that are to be subjected to relabelling. At present the n in the SELECT statement has no significance, since as many reels will be supplied as necessary for output, or read as input until end-of-file is reached.

CLOSE

When a program is to be run on multiprogramming machines, all slow peripheral files should be CLOSEd as soon as possible to release the units for other programs.

However, in the case of a non-multiprogramming installation, it would be better from the point of view of conserving storage to put all the CLOSE statements in a single segment (usually the End-of-Run segment). The same applies to OPEN statements, which go into a Housekeeping segment.

Magnetic tape files are not put off-line by the COBOL CLOSE statement. However, input files are released by MTH as soon as end-of-file is read, and in this case the tape *is* put off-line. The COBOL CLOSE referring to such files merely causes the fact that the file is closed to be noted in the MTH tables and the Run Time File table. If, therefore, it is required to read a tape several times during the running of an object program the tape will have to be put on-line by the operator each time the file is re-opened. The action required is merely pressing a button; the tape does not require re-threading.

This operator action can be avoided by writing a dummy marker block (or two such blocks if ALTERNATE AREAS are specified for the file) before the Trailer Label when the tape is created. Then when the file is read the program can test, after each READ, for the marker block, and when it is detected give a COBOL CLOSE. This will prevent MTH from reaching the Trailer Label and so putting the tape off-line.

ENTER

The following points should be noted:

(a) All literals, whether numeric or non-numeric, supplied as parameters to ENTERed routines, go into Upper Storage in the object program.

(b) For the address forms in which COBOL data fields are supplied as parameters to ENTERed routines see page 83.

(c) Care must be taken in referencing such fields: e.g. a PLAN STO instruction can only be used to reference a field occupying a single, whole word, and beginning on a word boundary (this may be checked from the Data Map listing of the COBOL source program).

(d) The order of ENTERed routines in semi-compiled form on a magnetic tape or paper tape is immaterial; if, however, these routines themselves CALL other routines held on the same tape, then the latter must *follow* the routines by which they are CALLed, otherwise they will not be consolidated into the COBOL object program.

(e) ENTERed routines that define and refer to their own magnetic tape files must number them *after* the magnetic tape files defined in the COBOL source program (see page 247), and in any case must not use the unit number 0.

(f) ENTERed routines can use the standard entry points 21 to 29, if the COBOL source is not segmented into overlays, or 23 to 29 if it is, subject to the restriction that they must not use any entry points required by any standard 1900 routines that they may call.

(g) If the ENTERed routines call MTH routines on the same tape as themselves, and these routines are also required by the COBOL object program, then only the versions on the subroutine tape will be consolidated, i.e. all required routines on the subroutine tape are consolidated before routines on the COBOL system tape.

(h) For the form in which COBOL procedure names are supplied as parameters to ENTERed routines see page 83; to use these in the subroutine it will usually be necessary to CALL one of the standard 1900 routines (e.g. %TROL) if the procedure names are in an overlay segment. If they are not, they may easily be referenced by means of the PLAN EXIT verb, for example. The ENTER statement generates CALL 1 (routine name) followed by the parameters as LDX or LDN 3 instructions.

(i) There are certain restrictions on the use of MTH in ENTERed routines, due to the fact that the compiler-generated object code for COBOL verbs OPEN, READ, WRITE and CLOSE uses its own Run-time File table (see page 244) to record and examine the state of the files.

When a file is OPENed a bit is set in its RTFT entry to indicate that it is open, and whether for input or output. Similarly when, on Reading, an End-of-File marker is met, a bit is set to indicate this fact, and that the file is not yet closed.

At run-time before any input/output instruction is performed the table entry for the file is examined to see if the instructions are permissible (e.g. a file cannot be closed unless previously opened, nor can it be read if end-of-file has already been reached). If they are not permissible the program will be suspended and an error message typed (see page 247).

It can be seen from this that it is perfectly possible to OPEN a file in COBOL, read or write it in an ENTERed routine, and then CLOSE it in COBOL. It would not, however, be possible to open it in an ENTERed routine and then READ or WRITE it in COBOL, because the object code for the latter verbs would find, from the Run-time File table, that the file had apparently not been OPENed. Consideration will also show the user what other 'mixtures' (of MTH usage in the COBOL program and in ENTERed routines) will be invalid.

PERFORM

The partitioning of a source program into subroutines which can be PERFORMed is of the highest value in conserving storage space. This applies not only when the PERFORM statement is in the same overlay as the subroutine to which it refers, but when the PERFORM is in Permanent, and the subroutine is in an overlay (for the rules governing PERFORMs see page 94).

It applies, too, not merely to the composing of a number of different statements into a subroutine, but to certain single COBOL statements which generate a large number of object instructions. A READ, for instance, generates between 25 and 35 instructions, depending on the peripheral (see pages 244 and 245 for the logic of these instructions). Then, if there are two or more READs of the same file and with the same AT END procedures in a segment, it is much more economical to label the READ statement with a paragraph name and PERFORM it at the appropriate points than to repeat the READ statement at each point.

A PERFORM generates five instructions, including two jumps; the above remarks therefore apply to all the input/output verbs, as well as to a number of other forms of statements, such as a MOVE to an edited target field.

When there are different exits from a PERFORMed procedure, e.g. via GO statements or SIZE ERROR and AT END branches, necessitating a separate EXIT paragraph, the verb referencing this procedure must be a PERFORM .... THRU ...., and not a simple PERFORM.

For example if there is a procedure:

```
PARA-A.
        MOVE X TO Y.
        ADD Z TO Y ON SIZE ERROR GO TO PARA-B.
        MOVE W TO Z.
PARA-B.
        EXIT.
```

It would be incorrect to write

```
PERFORM PARA-A.
```

because if a size error condition resulted from the ADD statement there would be a jump *outside* the PERFORMed range to PARA-B. It is necessary in such a case to write PERFORM PARA-A THRU PARA-B because then, whether there is a size error condition or not, control always returns to the instruction after the PERFORM statement.

Programmers should also make sure that if they jump out of a PERFORMed string, via GO statements, AT END or SIZE ERROR branches, they do eventually return to it, otherwise the result at run-time will be unpredictable, because the *link* has not been reset. Suppose, for instance, there is a statement PERFORM BONUS-CALCULATION, and elsewhere are these procedures:

```
BONUS-CALCULATION.
    MOVE ACTUAL-QTY TO WORK-1.
    MULTIPLY WORK-1 BY BONUS-RATE ON SIZE ERROR
    GO TO BONUS-CALC-ERROR.
OVERTIME-CALCULATION.
    ADD WORK-1 TO BASIC-PAY.
    .
    .
    . (etc.)
BONUS-CALC-ERROR.
    MOVE HEADER-3 TO PRINT-LINE.
    WRITE PRINT-LINE.
    MOVE MAXIMUM-QTY TO ACTUAL-QTY.
    .
    .
    .
    GO TO BONUS-CALCULATION.
```

Suppose then,

(a) that after BONUS-CALCULATION has been PERFORMed there is a size error, causing control to pass to BONUS-CALC-ERROR,

(b) that after BONUS-CALC-ERROR and other procedures it is intended, not to PERFORM the BONUS -CALCULATION paragraph but to 'fall through' it to the OVERTIME-CALCULATION paragraph.

This effect would *not* be achieved by the program as written, since after jumping out of the BONUS -CALCULATION paragraph on the size error the link at the end of the paragraph (generated by the PERFORM) will not be reset and so will still point at the statement after the PERFORM. Thus when the program tries to 'fall through' this paragraph it will instead return to the statement after the PERFORM.

There are of course circumstances in which the programmer might deliberately make use of this sequence; however, to achieve the effect described in (a) and (b) he might define an area called SWITCH -ERROR, initially set it to zero, and write:

```
    PERFORM BONUS-CALCULATION
    IF SWITCH-ERROR EQUAL TO 1 GO TO
    BONUS-CALC-ERROR.
    .
    .
    .
```

```
BONUS-CALCULATION.
    MOVE ACTUAL-QTY TO WORK-1.
    MULTIPLY WORK-1 BY BONUS-RATE ON SIZE
    ERROR MOVE 1 TO SWITCH-ERROR.
OVERTIME-CALCULATION.
    ADD WORK-1 TO BASIC-PAY.
    .
    .
    .
BONUS-CALC-ERROR.
    MOVE ZEROS TO SWITCH-ERROR.
    MOVE HEADER-3 TO PRINT-LINE.
    WRITE PRINT-LINE.
    MOVE MAXIMUM-QTY TO ACTUAL-QTY.
    .
    .
    .
    GO TO BONUS-CALCULATION.
```

(See pages 94 to 96 for all the PERFORM options implemented.)

STOP

It is not obligatory to have a STOP RUN in the source program. However, it may be recommended that there should be at least one STOP RUN statement, because

(a) if the operator tries to restart a program at object time, by causing the next instruction to be obeyed after a STOP RUN has been reached, the program will loop and suspend, displaying HALTED: END OF RUN each time the program is restarted,

(b) the coding generated for STOP RUN includes a test whether all files opened have been closed, and if the test fails, prints the cautionary message FILE nn NOT CLOSED AT EOR for each file unclosed.

In the case of branches to error stops, if the user wishes to continue the run he will usually follow the STOP statement with a GO back to whichever processing path he wishes to pick up again. (The next statement after the error STOP will always be obeyed at run-time, whatever it may be, if the operator restarts by typing GO.)

WRITE

When a printer output file is single buffered in the object program the object code generated is

```
(insert Print File Control Character in buffer)
PERI
SUSBY
```

When such a file is double buffered (i.e. its SELECT ... ASSIGN ... statement in the Environment Division includes the clause RESERVE 1) the object code generated is in effect

```
(insert PFCC in the 'other' buffer A, or B)
Move record from buffer B, or A
SUSBY
Switch the buffer pointer to address the 'other' buffer A, or B
PERI
```

Note: No peripherals in the object program, whether input or output, will be double buffered unless the user specifically orders it with the RESERVE 1 option.

WRITE (record-name[1]) FROM (record-name[2])

When this statement refers to a magnetic tape or direct access file (record-name[2]) must be defined with an initial word-count subfield, PICTURE 1(24) VALUE ZERO(E)(S), unless it is a record in a file defined as RECORDING MODE F (or V). If it is not, and if this subfield is not so defined the results at run-time will be unpredictable. The reason will be readily apparent if it is recalled that WRITE ... FROM ... is equivalent to the two COBOL statements

MOVE ...

WRITE ...

Chapter 13 Subroutines for use in COBOL programs

This chapter describes the means of incorporating various subroutines into COBOL programs. Information about these subroutines is given under the following headings:

Processing Direct Access Files by Subroutine

Sorting Magnetic Tape Files by Subroutine

Sorting Direct Access Files by Subroutine

Processing Variable -length Fields by Subroutine

PROCESSING DIRECT ACCESS FILES BY SUBROUTINE

A set of PLAN subroutines is provided to enable users to write COBOL programs processing files held on direct access devices. These subroutines can be used to process files which have been assigned to Exchangeable Disc Store, Fixed Disc Store or Magnetic Card File in the Environment Division. Each input or output operation is then written in the Procedure Division as an ENTER statement calling the subroutine appropriate in the function required and naming the file or one of its records as a parameter. Information relating to the use of these subroutines is given below under the following headings:

Introduction

Calling the Subroutine

Introduction

1 The COBOL user is referred to the *Direct Access* manual for information on the direct access hardware and on the software facilities that are provided. These PLAN subroutines call the Direct Access Housekeeping package at the logical level which means that the user must adhere to the information standards and the formats described in Chapter 16, (page 103), of *Direct Access*; it also means that both the Basic Software facilities and the Extended Software facilities are available to him.

2 In the description of a direct access file the clauses VALUE OF ID, ACTIVE-TIME, BLOCK CONTAINS, RESERVE ALTERNATE AREAS have the same effect as for a magnetic tape file in that they specify the label, retention period, buffer size and number of buffers respectively, to be assigned to the file. If the clause RESERVE 1 ALTERNATE AREA is written for a direct access file, one of the two buffers allocated will be used as the 'home' bucket buffer and the other as the 'overflow' bucket buffer.

3 The automatic run-time checks (e.g. that a file has been opened before being read or written to), which are normally carried out when input/output verbs are written in the source program, are not made when the ENTER verb is used. Any such illegal attempts at reading or writing are picked up by the Direct Access Housekeeping Package in the standard way. The format and significance of the Housekeeping error messages are described in *Direct Access*, page 127. It will be seen there that these messages ally the error condition to the appropriate file by quoting its file number. The direct access file numbers are the same as the numbers quoted in the SELECT statements for the files. The unit numbers quoted in this statement must be in the range 1 to 15. This does not limit the number of Direct Access files in a program, since any number may be declared with the same unit number. It is not possible, however, for more than one Direct Access file of a particular unit number, irrespective of its device, to be open at the same time (thus if files are ASSIGNed TO EDS 2, EDS 2 and FDS 2, only one may be open at any one time); the number that may be open simultaneously is therefore limited to 15.

4 Generation numbers are not automatically requested by the object program for files processed by ENTER verbs. If a generation number check is required the generation number must be set in a field with PICTURE S1(23) SYNCHRONIZED RIGHT. If this number is to be supplied to the program at run-time it must be input by means of an ACCEPT statement. This data area is then presented as one of the parameters to the subroutine which opens the file in question. If this field is negative when the OPEN

is initiated, the file with the highest generation number amongst all the files in the system with the specified label is the one that is opened. If it is positive the twelve least significant bits are taken as the generation number that is required. When opening an output file two generation numbers are required; the current one and the new one to be written to the file. The twelve least significant bits of the field specifying the new generation number are taken as the number required irrespective of whether it is positive or negative.

5 As in magnetic tape files a field with PICTURE 1(24) must be described at the beginning of each record of a direct access file. This is used by the Housekeeping system to hold the size of the record. In the case of direct access records, however, the twelve most signfiicant bits of the field are used to hold other information so that if the user wants to find the size of an input record from this field he must describe it as having two subfields with PICTURE 1(12); in this way he can access the twelve least significant bits. As far as writing records is concerned, these subroutines behave exactly as the COBOL WRITE verb operating on magnetic tape records. That is, unless the user inserts a non-zero value in the first field of the record, the size of the record as specified in the Data Division is inserted by the subroutine; otherwise the value that the user has inserted in the field is taken as the size of the record. The RECORDING MODE clause must not be used. Although OCCURS . . .DEPENDING can be used, the subroutines ignore it and assume the maximum record size.

6 In processing files held on direct access devices a number of exception conditions can arise. The identifier for a particular error condition is placed in words 10 and 11 of the program by the Housekeeping Package. The subroutines recognize the condition 'end-of-file marker encountered' and pass control to the procedure named by the end-of-file-label parameter of the subroutine which opened the file. If any other error condition arises, the contents of words 10 and 11 are set in the reply field named in the opening statement for the file and control is passed to the user's exception label. On identifying the error condition from the contents of the reply field, the user may choose to obey the statement ENTER PLAN CRETURN. Entry to this routine will cause control to be returned to the COBOL program at the statement following the one which caused the exception condition. The reply field is not automatically zeroized, but remains set at the identifier for the last exception condition encountered.

7 No dump facilities are supplied with the subroutines. It is not possible either to use a direct access file to control dumping, or to dump onto a direct access device.

Calling the subroutines

In the following descriptions of the COBOL statements which generate calls to the subroutines, the words printed in lower case in the formats represent parameters, whose values for the call the user defines by entering either a literal or a data name, as specified in the accompanying descriptions of the parameters.

OPENING A FILE

A file may be opened in any of the modes acceptable to the Housekeeping Package; as an input file, as an input-output file, as an output file, or at end of file. These modes of opening are described in *Direct Access,* page 100 (note that in *Direct Access*, input-output mode is called "overlay mode"). The mode of opening required is specified by the name of the subroutine in the ENTER statement, e.g. to open a file for input the subroutine COPENINPUT is involved.

Overflow

Before opening any file in output mode or in input-output mode, the user must define the highest level of overflow to be handled in the object program. This is achieved by entering the PLAN subroutine CLEVEL1 or CLEVEL2 as appropriate (see below). The level of overflow to be handled on each particular file is specified when opening the file. Thus, for example, if the first file to be opened for output is to have no overlay handling, the second has first level overflow and the third has second level overflow, then the program must call CLEVEL2 before opening the first file, and then call COPENOUTPUT or COPENINPUT to open the first file, giving the type-of-processing as "no overflow"; subsequently the second file will be opened with first level overflow and the third file will be opened with second level overflow. If no overflow is to be handled on any file, then neither CLEVEL1 nor CLEVEL2 need be entered. If neither CLEVEL1 nor CLEVEL2 is entered, then an error will arise if overflow arises on any file processed by the program, regardless of the level of overflow specified when opening the file. The error is exception condition # 25 (see *Direct Access*, page 129).

1 If the highest level of overflow in the program to be handled by Housekeeping is first level, the program must obey the statement

ENTER PLAN CLEVEL1

prior to the opening of any file requiring the facility. Exception # 25 will occur if second level overflow arises in a program having this statement.

2 If the highest level of overflow to be handled by Housekeeping is second level, the program must obey the statement

ENTER PLAN CLEVEL2 USING buffer-name

prior to opening any file requiring overflow handling.

Buffer-name is the name of an area in WORKING-STORAGE SECTION, whose size is defined by the user as the size of the largest record of all the files for which type-of-processing value 2 is specified. This area is not available to the user at any time that such a file is open. If the area is to be used by the program, given the above restriction, the writing of a complex data description will in no way affect the usage of the area by Housekeeping. Conversely, if the area is to be used exclusively by Housekeeping, it is not necessary to write a complex data description entry for it. For instance, the following would suffice:

77 BUFFER PIC X(120)

or, if the largest record were 124 characters:

01 OVERTHELIMIT.

02 FILLER PIC X (120).

02 FILLER PIC X (4).

Failure to obey the statement ENTER PLAN CLEVEL2 . . . before opening a particular file will cause exception # 25 if second level overflow arises on that file, regardless of the type-of-processing value.

Note that the ENTERing of CLEVEL2 implies that handling of both first and second level overflow is required and it is not then necessary to ENTER CLEVEL1.

File opening subroutines (other than scratch files)

In the following definitions, the meanings of the parameters are as follows:

File-name is the FD level name given to the file in Data Division.

Type-of-processing specifies the type of processing to be applied to the files. The values of this parameter are S, N, Q, 1 and 2; their significance is as follows:

Value	Significance
S or N	Serial processing. No overflow is to be handled by Housekeeping on this file; exception # 25 (see *Direct Access*, page 129) will occur if any overflow condition arises in processing this file.
Q or 1	Sequential or random processing: first level overflow only is to be handled by Housekeeping on this file. Exception # 25 will occur if a second level overflow condition arises in processing the file.
2	Sequential or random processing: first and second level overflow conditions on this file are to be handled by Housekeeping; exception condition # 25 will only arise if second level overflow occurs when the second level overflow.

Note: see also the section *Overflow* above.

Reply-field is the name of a group field which has two subfields, each with a picture of 1(24) SYNCHRONIZED RIGHT, in which reply information is set before exiting to the exception label. The first field will contain B0 to B5 of word 11 of the program, right justified; the second will contain the value of word 10, when this is applicable. (See *Direct Access*, pages 129 to 130, *Exception conditions*.)

Exception label is the name of the procedure to which control is to be passed when there arises any of the exception conditions (other than *end-of-file-label encountered*) that are listed in *Direct Access* pages 129 to 130, *Exception conditions*. This procedure may be an overlay of the object program.

Generation-number-field is the name of a field with PICTURE S1(23) SYNCHRONIZED RIGHT. Its significance is explained on page 149.

End-of-file-label is the name of the procedure (which may be an overlay of the object program) which is to be obeyed when the end-of-file marker is detected. Even if the end-of-file marker can never be detected in the processing of a file the parameter must still be specified. In this case, however, it can be either a numeric literal or the name of any procedure in the COBOL program.

The statements used to open the files are as follows:

1 To open a file for input:

ENTER PLAN COPENINPUT USING file-name, type-of-processing, reply-field, exception-label, generation-number-field, end-of-file-label.

2 To open an input-output file:

ENTER PLAN COPENINOUT USING file-name, type-of-processing, reply-field, exception-label, generation-number-field, end-of-file-label.

3 To open a file for output:

ENTER PLAN COPENOUTPUT USING file-name, type-of-processing, reply-field, exception-label, old-generation-number-field, new-generation-number

4 To open a file at end-of-file:

ENTER PLAN COPENEND USING file-name, type-of-processing, reply-field, exception-label, generation-number-field, end-of-file-label.

Scratch file opening subroutine

To open a scratch file on direct access:

ENTER PLAN COPSCRATCH USING file-name, S, reply-field, exception-label, scratch-inf, end-of-file-label.

Scratch-inf is the name of a group field with the following two subfields:

1 A field with PICTURE 9(4) which contains a value (which must be a multiple of 8) signifying the number of blocks required.

2 A field with PICTURE 9(6) which contains the serial number of the storage unit on which the scratch file is to be opened. If zero is entered, then the file is opened on any available storage unit.

The remaining parameters are identical in function to those of the other file opening subroutines. Note that the type-of-processing must be S, because scratch files can only be opened in serial mode.

The file will be deallocated automatically when it is closed, or when the program using it is deleted. Thus if the file is reopened the previous information is not accessible. Scratch files can, however, be extended, contracted and reset in the normal way.

Scratch files should be described in the Environment and Data Division in the usual way. The VALUE OF ID clause must be written, but the literal is not used and may have any value.

READING A FILE

1 To read the next record from a file:

ENTER PLAN CREAD USING file-name.

If the file is stored serially, then, irrespective of the type of processing specified to the open subroutine, repeated use of this subroutine presents the records in the order in which they were written to the file. If the file is stored sequentially, and opened for sequential processing, records are presented in the order of key sequence, i.e. tags are followed to overflow buckets. If it is stored sequentially and opened for serial processing, tags are not followed and tag records are not presented, i.e. records are presented in the order in which they are stored on the device. In the latter case some of the records in overflow buckets may not have been read by the time the end-of-file marker is encountered.

2 To read a specified record from a sequentially stored file: either

(a) ENTER PLAN CSREAD USING file-name, symbolic-key-name, or

(b) ENTER PLAN CSREADA USING file-name, symbolic-key-name, area1, area2, size1, size2.

To use these subroutines the file must be indexed in the standard way and must be opened for sequential processing.

Symbolic-key-name is the name of the field which holds the key of the required record.

If CSREAD is used the index tables are read into the file buffer every time the subroutine is obeyed. If the processing involves a lot of selective reading from the file it is more efficient to use CSREADA which

reads the index tables into Area1 and Area2 on the first access and assumes them in these areas subsequently.

Area1 and area2 are the names of two areas used at the user's discretion to contain the whole or a part of the corresponding level index table. Both the areas must begin on a word boundary.

Size1 and size2 are either the names of fields described with a Picture of 9(4) or numeric literals, each four characters in length. In either case their values specify the lengths in words of Area1 and Area2 respectively. These lengths must be three words longer than the storage required to hold the actual indices.

It is permissible to declare only one of the areas. In this case the area is specified by the parameters Area1 and Size1 in an ENTER statement for any of the subroutines CSREADA, CSINSERTA and CSDELETEA and the parameters Area2 and Size2 must be written as literal zero.

If, in processing a sequentially stored file, an ENTER CSREAD or an ENTER CSREADA is followed by a series of ENTER CREADs, then the records from the record specified in the ENTER CSREAD or ENTER CSREADA onwards, are presented in order of key sequence.

3 To read a specified record from a randomly stored file:

ENTER PLAN CRREAD USING file-name, symbolic-key-name, actual-key-name.

To use this subroutine the file must be randomly stored and must be opened for random or sequential processing. Before obeying each ENTER CRREAD the key-conversion routine associated with the file must be applied to the key of the required record. This routine transforms the key held in the field names in the symbolic-key-name parameter into a bucket number (actual-key) which it stores in the field named in the actual-key-name parameter.

Symbolic-key-name is the name of a character field of size equal to the key-length defined for the file.

Actual-key-name is the name of a field described as 1(24) SYNCHRONIZED RIGHT.

WRITING A RECORD

1 To write the next record to a file opened in output modes:

ENTER PLAN CWRITE USING record-name.

Record-name is the COBOL name of the record.

The output mode should only be used for creating serial files. All files written in the output mode are serial files as far as software is concerned even if the records are written in record key sequence. Sequential or random files should be loaded by the utility routines provided.

2 To replace the last record read of a file opened in input-output mode:

ENTER PLAN CREPLACE USING record-name.

To use this subroutine, the file, which may be serially, sequentially or randomly stored, must be opened in input-output mode for serial or sequential processing as appropriate.

If the replacement of a record in a file that is either serially stored or open for serial processing would cause bucket overflow, the transfer is not carried out and an exception condition arises.

3 To insert a record in a file before the last one read:

ENTER PLAN CWRITE USING record-name.

To use CWRITE with this effect, the file, which may be sequentially or randomly stored, must be opened in input-output mode for sequential processing. CWRITE must not be applied to a file with standard indexes. Insertions to such a file must be made by means of CSINSERT. The statement is invalid for a file that is either serially stored or open for serial processing, and will give rise to an exception condition.

4 To insert a record in sequence in a file: either

(a) ENTER PLAN CSINSERT USING record-name, symbolic-key-name, or

(b) ENTER PLAN CSINSERTA USING record-name, symbolic-key-name, area1, area2, size1, size2.

To use these subroutines the file must be sequentially stored, indexed in the standard way and opened in input-output mode for sequential processing. CSINSERT and CSINSERTA are related in exactly the same way as CSREAD and CSREADA which are described in note 2 of 'Reading a File'.

5 To insert a record in a randomly stored file:

ENTER PLAN CRINSERT USING record-name, symbolic-key-name, actual-key-name.

To use this subroutine the file must be opened in input-output mode for sequential processing. The other remarks and definitions in the description of CRREAD in note 3 of 'Reading a File' apply exactly to CRINSERT.

DELETING A RECORD

1 To delete the last record read of a file:

ENTER PLAN CDELETE USING record-name.

To use this subroutine the file, which may be stored serially, sequentially or randomly, must be opened in input-output mode for the appropriate type of processing.

2 To delete a specified record of a sequentially stored file: either

(a) ENTER PLAN CSDELETE USING record-name, symbolic-key-name, or

(b) ENTER PLAN CSDELETEA USING record-name, symbolic-key-name, area1, area2, size1, size2.

To use these subroutines the file must be indexed in the standard way and opened in input-output mode for sequential processing. Otherwise the remarks on CSREAD and CSINSERT apply also to these subroutines.

3 To delete a specified record of a randomly stored file:

ENTER PLAN CRDELETE USING record-name, symbolic-key-name, actual-key-name.

To use this subroutine the file must be opened in input-output mode for sequential processing. Except for those relating to the mode of opening, the remarks in CRREAD apply equally to CRDELETE.

EXTENDING A FILE

To extend a direct access file:

ENTER PLAN CEXTEND USING filename, number of blocks, storage unit serial number.

Number of blocks is either the name of a field with PICTURE 9(5) DISPLAY containing the number, or a numeric literal five characters in length. In either case the number of blocks must be a multiple of 8.

Storage unit serial number is either the name of a field with PICTURE 9(6) DISPLAY containing the number, or a numeric literal six characters in length. In either case the number is the serial number of the storage unit which is to contain the buckets of the additional file area created by the extension. If the value of the parameter is zero, the file is extended on any available storage unit.

CEXTEND will, if necessary, create more than one new file area in order to give the user the entire number-of-blocks specified in the parameter. Once ENTERed, the routine will proceed to create new file areas until either the number-of-blocks quoted has been obtained or until there is no space available for further extensions.

If the storage-unit-serial-number is quoted as other than zero, all the new file areas created by the ENTER will be on the unit number quoted.

The contents of the reply-field differ from those at present supplied by Housekeeping.

The first word of the reply-field is now subdivided into two parts:

Bits 0 to 17	Number of blocks given. This figure will always be correct, that is exactly the number of blocks allocated by Executive. If the extension was completed successfully, this figure will be not less than the number-of-blocks quoted in the parameters to the ENTER. If the extension was only partially successful, the contents of this field will be the number of blocks which it has been possible to allocate.
Bits 18 to 23	Exception code supplied by Housekeeping (see *Direct Access*, page 129).

The second word of the reply field will contain the last logical bucket number before extension, regardless of anything specified for word 10 in *Direct Access.*

CONTRACTING A FILE

To contract a direct access file:

ENTER PLAN CCONTRACT USING filename, number-of-blocks

Number-of-blocks is either a numeric literal five characters in length or the name of a data area having PICTURE 9(5) DISPLAY.

The effect of the above statement is to contract the file by an integral number of file areas, so that the number of blocks removed is as near as possible to, but never greater than, the number of blocks requested. File areas are removed from the end of the file.

For example, a file has three areas of 40, 80 and 40 blocks respectively. A contraction of 20 blocks would have no effect; a contraction of 80 blocks would only remove the last file area, that is, 40 blocks; and a contraction of 120 blocks would remove the complete 120 blocks of the last two areas.

RE-ATTEMPTING AN UNSUCCESSFUL ACCESS

To repeat an unsuccessful attempt to access a direct access file:

ENTER PLAN CREPEAT

The effect of this statement is that the program repeats the last attempted access of any direct access file to have caused an exception. Note that the term *exception* in this context includes the end-of-file condition. Users are warned to handle this facility with great care, because of the risk of creating infinite loops in object programs. For example, if overflow occurred on writing to a serial file, the user might choose to extend the file and then repeat the original write by ENTERing CREPEAT. However, if the extension caused an exception, the effect of the statement would be to repeat the abortive extension, not the original write.

The statement may appear in either permanent or overlay program. The statement which is to be repeated must be either in permanent program or in the same overlay as the ENTER PLAN CREPEAT statement.

CLOSING A FILE

To close a direct access file either:

ENTER PLAN CCLOSE USING file-name, or

ENTER PLAN CRESET USING file-name,

CCLOSE causes the file to be deleted from the program reservations.

CRESET causes the file to be retained by the program and to appear as if it had just been opened.

To return to the 'return address':

ENTER PLAN CRETURN

On obeying this subroutine after exiting to the exceptions label, control is passed to the COBOL program at the statement after that which caused the exception condition. This statement must either be in permanent program or in the same overlay as the ENTER PLAN CRETURN statement.

SORTING MAGNETIC TAPE FILES BY SUBROUTINE

The subroutine COBOLSORT will sort records input from a single simple magnetic tape file, the format of which conforms to I.C.T. 1900 Series magnetic tape organization standards. Information relating to the use of this subroutine is given under the following headings:

Input Data

Output Data

Hardware Requirements

Calling the Subroutine

Console Typewriter Messages

The explanation of the subroutine concludes with an example of a program using COBOLSORT.

Input Data

1 FILE FORMAT The input magnetic tape file must be recorded on a single reel of standard half-inch wide tape in odd parity mode and must conform to the I.C.T. 1900 Series magnetic tape organization standards regarding header labels, trailer labels, the various types of tape sentinels and record and block formats. The reel sequence number of the tape will not be checked. The file must be terminated with an end of file or end of reel trailer label. The file name must be described as a parameter.

Any information on the input magnetic tape reel which follows the header label and precedes the start of data sentinel or which is enclosed by standard tape sentinels is ignored and will not be output. The subroutine recognizes only an end of file or end of reel trailer label as the end of the input data.

2 RECORD FORMAT Records may be of fixed or variable length and may be batched or unbatched. They must be at least two words but no more than 512 words in length.

The first word in each record must contain a count of the number of words in the record.

3 BLOCK FORMAT Blocks may contain one or more records. They must be at least five words but no more than 1,024 words in length. When the record information in a block occupies less than five words the word following the final word of record information must contain zero. The contents of other words are immaterial and will be ignored. If the first word of any record in a block contains zero that record and all subsequent information in the block will be ignored.

4 KEYS Those portions of a record by which it is to be referenced for sorting purposes are described as the *keys*, to which the following rules apply:

(a) There may be from 1 to 6 keys in a record but the number must be the same in each record.

(b) The order of priority of keys need not correspond to their order within a record.

(c) Records may be sorted on a key in either ascending or descending sequence irrespective of the order of sorting on other keys present.

(d) Each key must be fixed in length and must be continuous within a record.

(e) Each key must be in a fixed position relative to the start of a record.

(f) Each key specified must be complete and present in each record.

(g) Irrespective of the type of any other key present in a record a key may be of one of the following types:

(Types 0 and 1 are not available at present and will be ignored if used).

Type 0: An unsigned bit string of from 1 to 48 bits in length, starting at any bit position within a record.

Type 1: A signed bit string of from 1 to 48 bits in length, starting at any bit position within a record. Negative numbers are held in true complementary form.

Type 2: An unsigned string of from 1 to 120 six-bit alphanumeric characters starting at any character position within a record.

Type 3: An unsigned string of from 1 to 120 six-bit numeric characters starting at any character position within a record. For the purpose of key comparison a 'space' character (010000) is treated as being equivalent to a 'zero' character (000000).

Type 4: A signed string of from 1 to 120 six-bit characters starting at any character position within a record and having a separate 'sign' character. For the purpose of key comparison a 'space' character (010000) is treated as being equivalent to a 'zero' character (000000). If the first character of the string is a 'minus' character (011101) it is treated as being equivalent to a 'zero' character (000000) amd the key is considered to be negative. If the first character is not a 'minus' it is treated as being equivalent to a 'zero' (000000) and the key is considered to be positive.

Type 5: A signed string of from 1 to 120 six-bit characters starting at any character position within the record and having the sign represented by overpunching. For the purpose of key comparison a 'space' character (010000) is treated as being equivalent to a 'zero' character (000000). If the first character of the string is a 'minus' (011101) or a 'quotes' character (010010) it is treated as being equivalent to a 'zero' (000000) and the key is considered to be negative. Moreover, if the first character is J,K,L,M,N,O,P,Q or R it is treated as being equivalent to 1,2,3,4,5,6,7,8 or 9 respectively and the key is considered to be negative.

Note: All characters in keys of type 3, 4 or 5 are sorted only on the least significant four bits except in the cases specifically referred to already.

(h) Keys may overlap in any manner.

(i) The first word of a record, i.e. the word indicating the record size may be used as a key or as part of one.

(j) The comparison of signed numeric keys is algebraic, i.e. positive numbers are greater than negative numbers and small negative numbers are greater than large negative numbers.

Output data

1 FILE AND REEL FORMAT The output magnetic tape file will be recorded on standard half-inch wide tape in odd parity mode and will conform to the I.C.T. 1900 Series magnetic tape organization standards regarding header label, start of data sentinels and trailer labels.

The first reel in the file will be numbered Reel 0. Subsequent continuation reels will have identical header labels regarding file name, generation number and retention period as the first reel of the file except that they will be numbered consecutively from Reel 0. The security indicator bit will be set at zero on all reels. The file name, generation number and retention period of the file must be supplied to the subroutine as parameter information (see page 156).

2 RECORD FORMAT This is the same as for input.

3 BLOCK FORMAT Blocks must be as long as the longest record but must not exceed 1,024 words in length. The maximum block size must be supplied to the subroutine as a parameter. Records will be batched if possible.

4 SENTINEL INFORMATION The contents of Words 4 to 19 of the start of data sentinel of the input file will appear in Words 4 to 19 in the start of data sentinel of the output file. The contents of Words 4 to 19 of the end of file trailer label of the input file will appear in Words 4 to 19 of each trailer label of the output file.

Hardware requirements

1 MAGNETIC TAPE DECKS At least three and not more than sixteen magnetic tape decks must be available to the subroutine at the time of entry to it. The number available must be supplied to the subroutine as a parameter.

The input file must be in a closed state immediately before entry to the subroutine. During the sorting process all work tapes will be opened and labelled with a one-day retention period.

2 CONSOLE TYPEWRITER A console typewriter must be available to the subroutine.

3 STORAGE SPACE AND MAGNETIC TAPE DECK UTILISATION The subroutine requires 2,000 words of core store for lower data areas and the program of which 500 words are required for lower data. The actual amount of core store required for sorting is calculated by the subroutine and requested at run-time.

The amount of extra store required must satisfy the following two inequalities:

$$C > 2N(M + 5)$$

$$C > (2 \left[\frac{M}{R}\right] + 1)\ (1 + \sum_{z=1}^{z=6} \left[\frac{M_z + 3}{4}\right]) + 2(M + 5) + 1054$$

where

C is the total number of extra core store words required.

N is the number of magnetic tape decks to be used.

M is the maximum block size of the input or output files.

R is the minimum record length in words needed to contain all the keys specified. (This is not necessarily the absolute minimum record size.)

M_z is the number of characters in the zth key

The quantities in square brackets are rounded down.

(The sigma sign, Σ, indicates the sum of the values of the subsequent expression in which one item varies with a fixed range, in this case the number of characters in each of the keys specified.)

The subroutine first calculates the amount of storage necessary to effect the sort on the number of magnetic tape decks specified by the user. If this amount is not available, the subroutine finds out how many decks it can use with the available storage. If the number of decks is less than three, it displays an error message and exits.

Having decided how many decks it can use, COBOLSORT checks that this number is actually available for allocation. If it is not, the subroutine uses as many decks as are available or, if less than three decks are available, exits with an error message. If the sort can proceed, the number of decks in use is displayed on the console typewriter.

After the input tape has been read the message "REMOVE INPUT REEL FROM UNIT NN" is displayed. If the input tape has a write permit ring and no retention period, it is reopened as a work tape; otherwise the input tape must be replaced with a scratch tape, unless one is available on a hitherto unallocated deck.

If for any reason the user wishes to abandon the run, he can do so by typing "GO # program-name 25". This has the effect of scratching all the tapes used by the subroutine for sorting.

Calling the subroutine

To call the subroutine COBOLSORT the following COBOL statement must be used:

ENTER PLAN COBOLSORT USING file-name-1, file-name-2, record-name-1, record-name-2.

All four parameters *must* be present since no check is made and the action of the subroutine is undefined if any are missing. The parameters are as follows:

1 *File-name-1* is the file name of the input magnetic tape file described in the Data Division.

2 *File-name-2* is the file name of the output magnetic tape file described in the Data Division.

3 *Record-name-1* is a record described in the Working-Storage Section. It has the following format:

```
01 Record-name-1.
02 Number-of-tape-decks PICTURE 9999 VALUE N.
02 Input-file-generation-number PICTURE 1 (24) VALUE m.
02 Output-file-generation-number PICTURE 1(24) VALUE n.
02 Reply-word PICTURE 9 COMPUTATIONAL SYNCHRONIZED RIGHT.
```

N is the maximum number of magnetic tape decks to be used and must be at least 3 and no more than 16. The input and output file generation numbers must be supplied in this record for the time being.

On return from the subroutine the value of the reply word will be:

0 if sorting has been successful.

1 if the parameter information was incorrect or inconsistent.

2 if the file or record information was inconsistent with either the I.C.T. 1900 Series Magnetic Tape Housekeeping standards or with the parameter information or with both.

3 if there was insufficient storage available or insufficient magnetic tape decks available for sorting or both.

4 if RESTART POINT 25 (see below, Page 158) has been used, in which case all tapes will have been released and work tapes scratched.

4 *Record-name-2* is a record described in the Working-Storage Section. It contains information about the keys to be used and has the following format:

```
01 Record-name-2.
02 Number-of-keys PICTURE 9999 VALUE M.
```

01 Record-name-2.

02 Number-of-keys PICTURE 9999 VALUE M.

02 Direction PICTURE 1(6) VALUE a.
02 Key type PICTURE 1(3) VALUE b.
02 Key length PICTURE 1(15) VALUE c.
02 Key character PICTURE 1(9) VALUE d.
02 Key word PICTURE 1(15) VALUE e.

} These five fields occur M times

M is the number of keys specified and must be in the range 1 to 6.

a is the direction indicator and is

0 for ascending order

1 for descending order

b is the type number of the key in the range 2 to 5.

c is the length in characters of the key, including the sign character if any. It must be in the range 1 to 120.

d is the character position within a word of the first character of the key and must be within the range 0 to 3.

e is the word address within the record of the word containing the first character of the key. (The word containing the size of the record is counted as Word 0.)

If convenient and desired the five 02 levels for a, b, c and d and e can be combined. For example, a and b can be combined into one field of PICTURE 1(9) with a value in the range 0 to 5 for ascending order or 8 to 13 for descending order.

Console typewriter messages

The following messages may occur during the use of the subroutine COBOLSORT. The program does not halt because of any of them but displays the appropriate message, sets the reply word and exits to the statement following the ENTER statement. The reasons for the occurrence of the messages and the action taken by the subroutine are given below.

Message	Cause	Subroutines action
INCORRECT FILE LAYOUT	Either (a) the format of the input file is incorrect, or (b) one of the records exceeds the maximum data block length, or (c) one of the records is not long enough to contain all the keys specified.	Sorting is terminated and the reply word is set to 2.
BLOCKCOUNT ERROR	The data block count of the input file is incorrect.	Sorting is terminated and the reply word is set to 2.
REMOVE INPUT REEL FROM UNIT NN	The input file has been read and may be removed if necessary.	
NO RECORDS PRESENT ON SOURCE TAPE	The input file consists of a start of data sentinel and an end of data sentinel only.	Sorting is terminated and the reply word is set to 2.
INCORRECT PARAMETER	The parameters supplied are incorrect.	Sorting is terminated and the reply word is set to 1.

Message	Cause	Compiler action
BINARY KEY ILLEGAL, KEY IGNORED	Sorting has been requested on a binary key which is not permissible and the key has therefore been ignored.	Sorting continues unless no other key is available, in which case the reply word is set to 1.
FIRST O/P REEL ON UNIT XX FULL	The first reel of the sorted file on unit XX is full and a second reel is required for output.	
END OF SORT, FINAL O/P REEL ON UNIT XX	The sort has been completed with the final reel of the sorted file on unit XX.	Control passes to the statement following the ENTER statement in the main program.
WRITING XX	Unit XX is the unit being written to on the current merge.	
NN TAPE DECKS USED	NN magnetic tape decks are being used for sorting.	
INSUFFICIENT STORE AVAILABLE	Insufficient storage space is available for sorting on three tape decks.	Sorting is terminated and the reply word is set to 3.
INSUFFICIENT TAPE DECKS AVAILABLE	Less than three tape decks are available for sorting.	Sorting is terminated and the reply word is set to 3.
RUN ABANDONED	Restart Point 25 has been used and all tapes have been released.	The reply word is set to 4.

RESTART POINT 25

If there is a hardware failure, or sorting is terminated because of an error while all tapes are still allocated, and it is required to release all the tapes being used by the subroutine and scratch all work tapes, the operator must type GO # NAME 25.

Example program using COBOLSORT

THE PROBLEM

The records of a master file held on magnetic tape contain two keys, a name and a job-number. Before processing, the records are held in ascending order of job-numbers and alphabetical order of names within each job-number. It is required to do some processing on the file whilst in this order and then to sort the records into alphabetical order of names and descending order of job-numbers within each name. The new file is then to be processed.

THE COBOL STATEMENTS

The COBOL statements to effect the necessary sorting would be as follows in each of the Divisions affected:

1 Environment Division

```
SELECT UNSORTED-FILE ASSIGN 1 TAPES.
SELECT SORTED-FILE ASSIGN 1 TAPES.
```

2 Data Division

```
FILE SECTION.
FD UNSORTED-FILE LABEL RECORDS STANDARD WITH GENERATION-NO VALUE OF
ID "USF".
01 RECORD-1.
02 COUNT PICTURE 9(4) COMPUTATIONAL SYNCHRONIZED RIGHT.
```

02 JOB-NO PICTURE 9(5).

02 NAME PICTURE X(13).

' ' '

' ' '

Other information

01 RECORD-2.

02 COUNT
02 JOB-NO
02 NAME

} These fields must have the same form as the corresponding fields of RECORD-1.

' '

' '

' '

Other information

FD SORTED-FILE LABEL RECORDS STANDARD WITH GENERATION-NO VALUE OF ID "SF".

The record description must correspond to the records of UNSORTED-FILE.

WORKING-STORAGE SECTION.

01 KEY-RECORD-1.

02 TAPE-DECKS PICTURE 9(4) VALUE 5.

02 INPUT-GENERATION PICTURE 1(24) VALUE 2.

02 OUTPUT-GENERATION PICTURE 1(24) VALUE 3.

02 REPLY-WORD PICTURE 9 COMPUTATIONAL SYNCHRONIZED RIGHT.

01 KEY-RECORD-2.

02 NUMBER-OF-KEYS PICTURE 9(4) VALUE 2.

02 DIRECTION PICTURE 1(6) VALUE 0.

02 KEY-TYPE PICTURE 1(3) VALUE 2.

02 KEY-LENGTH PICTURE 1(15) VALUE 13.

02 KEY-CHAR PICTURE 1(9) VALUE 1.

02 KEY-WORD PICTURE 1(15) VALUE 2.

02 DIRECTION-1 PICTURE 1(6) VALUE 1.

02 KEY-TYPE-1 PICTURE 1(3) VALUE 3.

02 KEY-LENGTH-1 PICTURE 1(15) VALUE 5.

02 KEY-CHAR-1 PICTURE 1(9) VALUE 0.

02 KEY-WORD-1 PICTURE 1(15) VALUE 1.

Notes:

(a) The key information can be written as one field. For example, the second key above could be described by the following entry:

02 SECOND-KEY SIZE 8 VALUE "1(050001"

(b) The order of sorting need not correspond to the order of the keys within the record, as the example shows.

3 Procedure Division

OPEN INPUT UNSORTED-FILE.

Processing of UNSORTED-FILE.

CLOSE UNSORTED-FILE.

ACCEPT INPUT-GENERATION.

ACCEPT OUTPUT-GENERATION.

ENTER PLAN COBOLSORT USING UNSORTED-FILE, SORTED-FILE, KEY-RECORD-1, KEY-RECORD-2.

OPEN INPUT SORTED-FILE.

Processing of SORTED-FILE.

CLOSE SORTED-FILE.

STOP RUN.

Notes:

(a) Neither the input nor the output file may be open at the time of entry to COBOLSORT.

(b) If it is desired to supply generation numbers at run time, they must be put into the appropriate words of the parameters by means of ACCEPT statements.

An ACCEPT FILE KEYS statement is not sufficient. It follows therefore that if a check is required of the generation numbers of the input and output files when there are no file-keys corresponding to them, up to four console messages may be output. These are the normal 'ACCEPT . . . GENERATION NO' messages for each file when being opened and a 'COBOL ACCEPT . . .' for each file when the user is about to ENTER COBOLSORT.

SORTING DIRECT ACCESS FILES BY SUBROUTINE

A set of PLAN subroutines is provided to enable users to write COBOL programs sorting files held on direct access devices. Communication is by way of the ENTER statement in the Procedure Division. The complete system consists of two sets of subroutines and the sort proper (XSXA).

The constituent parts of the system

THE SUBROUTINES

1 The user ENTERed routines

CSORT

CSORTA

CSORTB

CSORTC

CSORTD

CSORTE

CPERM

2 The core resident routines

GCBLDASORT

GCBLDASORT1

GCBLDASORT2

GCBLDASORT3

GCBLHSKPNG

GCBLCOBDUMP

GCBLMTSURCH

GCBLMTLOAD

GCBLDASURCH

GCBLDALOAD

GCBLCOBOLLD

GCBLINBATCH

GCBLOUTBTCH

These subroutines should be put into the subroutines group (or groups) which will be searched by the COBOL compiler in question. Group 1 must precede group 2.

Of the group 2 routines, the first four routines and the last two (which control input and output procedures) will be incorporated into the COBOL program according to the group 1 routine(s) used, and the other seven will always be incorporated.

THE SORT PROPER (XSXA)

XSXA is a non-overlay program which is required to be in a library file (MT or direct access).

When the group 2 routines are entered via group 1, the COBOL program is dumped to the work files and XSXA is read in from its library file as if it were coming from the COBOL program's overlay file; when the sort is finished, the COBOL program is reloaded and entered accordingly.

General

1 INPUT. Only one input file is allowed, which can be serial, sequential or random (and must conform to Housekeeping formats). Only a complete file may be sorted (selection of individual file areas for sorting is not allowed). However, intead of an input file, the user may use an input procedure to present records to the sort one by one (see GCBLDASORT4, and GCBLDASORT5, *Cues within input and output procedures*, page 160.4).

2 OUTPUT. Only one output file is allowed, which will be written as a serial file following Housekeeping formats. Instead of an output file, the user may use an output procedure to present records to him one by one (see GCBLDASORT6, *Cues within input and output procedures*, page 160.4).

3 WORK FILES. Two work files are required, which must be named files (see CSORT). They will be on the same type of device as the input and output files (if present), and must have the same bucket size as the output file.

The following combinations of input and output are available:

1 Input file and output file: see CSORTA.

2 Input procedure and output file: see CSORTB.

3 Input file and output procedure: see CSORTC.

4 Input procedure and output procedure: see CSORTD.

File sharing is permitted, with the limitation that the input file cannot be shared with the first work file.

Keys

There may be up to 10 keys and these may overlap in any manner. Keys must be fixed length and must appear in fixed position in the record. The order of sorting on any key may be ascending or descending, irrespective of the order on other keys. For all numeric keys, sorting is algebraic.

Keys may only be of the following types:

1 UNSIGNED BINARY STRING. This type of key may be up to 48 bits in length, and may begin on any bit within a word. The 24th and 48th bits from the right hand end of the key are assumed to be non-significant.

2 SIGNED BINARY STRING. This type of key may be up to 48 bits in length and may begin on any bit within a word. The 24th bit from the right hand end of the key is assumed to be non-significant unless the key is 24 bits long. Negative numbers are held as binary complements.

3 ALPHANUMERIC CHARACTERS. This type of key may be up to 120 characters in length, starting on any character position. Any of the 64 6 bit characters may appear. Sorting is in accordance with the 1900 collating sequence.

4 UNSIGNED NUMERIC CHARACTERS. Keys of this type may be up to 120 characters long and may start on any character position. The only allowable characters are 0-9 and space. Space will be considered by the sort to be equal to zero.

5 SIGNED NUMERIC CHARACTERS (OVERPUNCH). This type of key is the same as type 4 except that the first character may contain an overpunch minus sign. In other words, the first character of a type 5 key may only be 0-9, J-R, space, '', or -. Subsequent characters must be 0-9 or space.

6 SIGNED NUMERIC CHARACTERS (SEPARATE SIGN). Keys of this type are the same as type 4 except that the first character will contain a sign. This may be space (positive), + (positive), or – (negative). Subsequent characters must be 0-9 or space.

Calling the subroutines

NOTES ON PARAMETERS

Wherever a generation number appears as a parameter to a subroutine, it must be the name of a field with PICTURE S1(23) SYNCHRONIZED RIGHT. This field is expected to contain the required generation number on entry to the sort. (Its value may be ACCEPTed etc.) If this field is positive, the twelve least significant bits are taken as the generation number. A negative value for a direct access file has its conventional meaning.

Procedure-name may be the name of a procedure in an overlay of the object program. This may be a different overlay from the one containing the sort.

File-name, for files described in the Data Division refers to the name in the SELECT . . . ASSIGN statement.

CSORT: THE SORT ENVIRONMENT DEFINITION

ENTER PLAN CSORT USING file-name-1, fgn-1, device-type, file-name-2, fgn-2, file-name-3, fgn-3.

file-name-1 is an alphanumeric literal (or zero), being the name of the library file containing XSXA; *fgn-1* is its generation number, and *device-type* gives the device on which it resides, as follows:

1 = E.D.S., 2 = Twin E.D.S., 3 = F.D.S.,

4 = Magnetic tape

If *file-name-1* is zero, then the other two parameters are ignored (but must be present) and the library file is assumed to be PROGRAM$_{\nabla}$XSXA generation number 0 residing on E.D.S.

File-name-2 is an alphanumeric literal (or zero), being the name of the first work file; *fgn-2* is its generation number. If *file-name-2* is zero, then *fgn-2* is ignored and the work file is assumed to be COB$_{\nabla}$WORKFILE generation number 1.

File-name-3 is an alphanumeric literal (or zero), being the name of the second work file; *fgn-3* is its generation number. If *file-name-3* is zero, then *fgn-3* is ignored and the work file is assumed to be COB$_{\nabla}$WORKFILE generation number 2.

Initially the system assumes that all three default options are in use. CSORT is only used when the environment is to be changed (e.g. if two successive sorts have the same environment it is not necessary to enter CSORT before the second of them).

CSORTA: SORT WITH INPUT FILE AND OUTPUT FILE

ENTER PLAN CSORTA USING file-name-1, fgn-1, file-name-2, fgn-2, record-name.

File-name-1 and *fgn-1* are the name and generation number of the input direct access file described in the Data Division.

File-name-2 and *fgn-2* are the name and generation number of the output direct access file described in the Data Division.

Record-name is the name of a record described in Working-Storage section. It has the following contents:

Words 1-2	Zero.
Word 3	Device type of sort (that is whereabouts of work, input and output files).
	1 = E.D.S., 2 = Twin E.D.S., 3 = F.D.S.
Words 4-7	Zero
	(Word 6 is used as a reply word: see *Restart point 29*).
Word 8	Number of keys (in binary).

The rest of the area is variable length, there being a two word entry for each key. These two words must hold:

Word 1	Character 0	1-6 = type of key	
	Character 1	0 = ascending key	
		1 = descending key	
	Character 2 and 3	Length of key in binary, in bits if key is type 1 or 2, in characters if key is type 3-6.	
Word 2	Start address of key (relative to beginning of record).		
	Key type 1 or 2:	B 0-4	bit address
		B 12-23	word address
	Key type 3 to 6:	B 0-1	character address
		B 12 - 33	word address

CSORTB: SORT WITH INPUT PROCEDURE AND OUTPUT FILE

ENTER PLAN CSORTB USING procedure-name, file-name, fgn, record-name, area-1, area-2

Procedure-name is the name of the procedure that is used to present records to the sort. (Section name or paragraph name in the Procedure Division.)

File-name and *fgn* are the name and generation number of the output direct access file described in the Data Division.

Record-name is as in CSORTA.

Area-1 is the name of the area in which the input procedure set up records. It must begin on a word boundary.

Area-2 is the name of an area to be used as a buffer during the input procedure. It must begin on a word boundary and be large enough to hold a work file bucket.

CSORTC: SORT WITH INPUT FILE AND OUTPUT PROCEDURE

ENTER PLAN CSORTC USING file-name, fgn, procedure-name, record-name, area-1, area-2.

File-name and *fgn* are the name and generation number of the input direct access file described in the Data Division.

Procedure-name is the name of the procedure used to accept records from the sort.

Record-name is as for CSORTA.

Area-1 is the name of the area in which records are presented to the user within the output procedure. It must begin on a word boundary.

Area-2 is as for CSORTB except that it is used during the output procedure.

CSORTD: SORT WITH INPUT PROCEDURE AND OUTPUT PROCEDURE

ENTER PLAN CSORTD USING procedure-name-1, procedure-name-2, record-name, area-1, area-2.

Procedure-name-1 is as for *procedure-name* in CSORTB.

Procedure-name-2 is as for *procedure-name* in CSORTC.

Record-name is as for CSORTA.

Area-1 is used for both purposes described in CSORTB and CSORTC.

Area-2 is as for CSORTB *and* CSORTC.

CSORTE: TO RETURN TO THE COBOL STATEMENT FOLLOWING THE SORT ENTER PLAN CSORTE

This routine, which may only be used in non-overlay programs, causes control be passed to the COBOL statement following the last executed CSORTA/B/C/D.

It will normally be used in the output procedure when it is not desired to process the sorted file through to the end.

CPERM

When a program is non-overlay and contains input and/or output procedures

ENTER PLAN CPERM

must be included, but never obeyed.

CUES WITHIN INPUT AND OUTPUT PROCEDURES

1 GCBLDASORT4: To present a record to the sort

ENTER PLAN GCBLDASORT4

When the input procedure has set up a record in *area-1*, this routine is used to present it to the sort.

2 GCBLDASORT5: To enter the sort at the end of the input procedure

ENTER PLAN GCBLDASORT5

When all the records have been presented to the sort, this routine is used to enter the sort proper.

3 GCBLDASORT6: To obtain a record from the sort

ENTER PLAN GCBLDASORT6 USING *procedure-name.*

This routine will present the next record of the sorted file to the user (in *area-1*).

Procedure-name is the name of the procedure to be branched to when the end of the file is reached.

Note: GCBLDASORT4/5 are cues within GCBLINBATCH, GCBLDASORT6 is a cue within GCBLOUTBTCH

Exception conditions

All exception conditions found by the sort will be indicated by a message on the console typewriter. This message will consist of the words SORT HALT or SORT ERROR followed by a two-character code. SORT HALT implies that the program may be able to continue after operator action. SORT ERROR implies that the program cannot continue.

The error codes are listed in Chapter 17.

Notes

It is not permitted to use either the VALUE OF ID IS *data-name*, or the ACTIVE-TIME IS *data-name* facility in the FD description for either the input or output file to the sort (when present). (The output file is opened in output mode during the sort, and will be given a retention period according to the ACTIVE-TIME clause.)

The input and output files (if present) and the work files must not be open on entry to the sort.

The sort will search for two unit numbers which are not in use (at time of entry) by the COBOL program, and will open its files under these numbers. The method of searching for unit numbers will be to start at 15 and work downwards. All files opened by the sort will be closed again before control is returned to the COBOL program, unless an output procedure is present when one file will be retained until termination of the sort.

With the above in mind, the user must ensure that no files with unit numbers used by the sort are opened within input and/or output procedures.

The sort requires a minimum of 5888 words of core store. If this much is not allocated to the COBOL object program, the sort will attempt to allocate itself the extra core needed. If 5888 words are not available, then the message SORT ERROR R1 will be produced on the console.

If the output file is unable to hold all the data, it will be extended by an amount sufficient to hold the remainder. The sort will endeavour to make this extension on the cartridge containing the last file area of the file.

The size of the work files should be carefully calculated. There must be room not only for the input data but also a further forty blocks to hold the merge and the generated key rearrangement routine and sufficient additional space to accommodate the user's program. It should be noted that it is the core allocated to the user's program at run time which is written to the work file, not necessarily the compiled allocation, for example, with a single programming executive the whole core store will be written.

The workfiles will not be extended if they are too small.

Example program with input and output procedures

THE PROBLEM

See page 158 for comparison with COBOLSORT.

The records of a master file held on magnetic tape contain two keys, a name and a job-number. Before processing, the records are held in ascending order of job-numbers and alphabetical order of names within each job-number. It is required to do some processing of the file whilst in this order and then to sort the records (using disc workfiles) into alphabetical order of names and descending order of job-numbers within each name. The new file is then to be processed, although a permanent copy of it is not required.

An input procedure will be used to present the magnetic tape records to the sort, and an output procedure will be used to present the sorted records for processing.

The sort is in PROGRAM$_{\nabla}$XSXA generation number 0 on E.D.S.

The 1 block/bucket work files are COB$_{\nabla}$WORKFILE generation numbers 1 and 3.

THE RELEVANT COBOL STATEMENTS

1 Environment Division

```
SELECT UNSORTED-FILE ASSIGN 1 TAPES.
```

2 Data Division

```
FILE SECTION.
FD UNSORTED-FILE LABEL RECORDS STANDARD WITH GENERATION-NO VALUE OF ID "USF".
01 RECORD-1.
02 COUNT PICTURE 9(4) COMPUTATIONAL SYNCHRONIZED RIGHT.
02 JOB-NO PICTURE 9(5).
02 NAME PICTURE X(13).
WORKING-STORAGE SECTION.
01 REC.
02 WDS12 PICTURE 9(8) VALUE ZEROS.
02 WD3 PICTURE 9(4) VALUE 1.
02 WDS45 PICTURE 9(8) VALUE ZEROS.
02 WD6 PICTURE 9(4) VALUE ZEROS.
02 WD7 PICTURE 9(4) VALUE ZEROS.
02 WD8 PICTURE 1(24) VALUE 2.
02 KEY1.
03 PT1A PICTURE 99 VALUE 30.
03 PT1B PICTURE 1(12) VALUE 13.
03 PT2A PICTURE 1(2) VALUE 1.
03 PT2B PICTURE 1(22) VALUE 2.
02 KEY2.
03 PART-1 PICTURE 9(4) VALUE 4105.
03 PART-2 PICTURE 9(4) VALUE 1.
```

Note that the key information can be written in more than one way.

```
01 BUFFER.
02 DESC PICTURE X(64) OCCURS 8 TIMES.
77 FGN PICTURE S1(23) SYNCHRONIZED RIGHT VALUE 3.
```

```
3  Procedure Division
   BEGIN.
       OPEN INPUT UNSORTED-FILE.
       ENTER PLAN CSORT USING 0, 0, 0, 0, 0,
       "COB_7WORKFILE",FGN,
       ENTER PLAN CSORTD USING INPROC, OUTPROC,
       REC, RECORD-1, BUFFER.
       CLOSE UNSORTED-FILE.
       STOP RUN.
       ENTER PLAN CPERM.
   INPROC.
       READ UNSORTED-FILE AT END GO TO TERM.
       Process record.
       ENTER PLAN GCBLDASORT4.
       GO TO INPROC.
   TERM.
       ENTER PLAN GCBLDASORT5.
   OUTPROC.
       ENTER PLAN GCBLDASORT6 USING FIN.
       Process record.
       GO TO OUTPROC.
   FIN.
       ENTER PLAN CSORTE.
```

PROCESSING VARIABLE LENGTH FIELDS BY SUBROUTINE

The four subroutines COBDIST, COBDISTM, COBPAK and COBPAKM provide an easy method of handling variable length input and output data fields held on paper and magnetic tape. Users can deal with variable length fields without using the subroutines but this involves a great deal of explicit coding which is eliminated by using the subroutines. Information relating to the use of these subroutines is given under the following headings:

Introduction

Calling the Subroutines

Key formats

Error Reports

The explanation of the subroutines concludes with examples of programs using them.

Introduction

COBDIST and COBDISTM are used to unpack a string of variable length items demarcated by field-terminators and possibly interspersed with fixed length fields into a series of consecutive fixed length fields in store. In the process the field terminators are omitted; the items are justified and space-filled according to keys specified by the user which generally correspond to the description of the receiving fields given in the Data Division.

COBPAK and COBPAKM have the opposite function. They convert a string of fixed length input data into a string of variable length output data.

Note that although it is stated that the subroutines handle input and output data, it should be clear that they do not perform any peripheral transfers. Their function is to unpack data immediately after input and to pack data immediately before output.

The two unpacking subroutines, COBDIST and COBDISTM, are identical except that COBDIST, which is primarily intended for data input on paper tape, requires a newline character (upwards arrow, asterisk ↑*) to terminate a record of variable length data. COBDISTM is primarily intended for use with magnetic tape on which the size of the record is normally determined by a word count of the number of words in the record. Correspondingly COBPAK moves a newline character into the receiving area after the last field packed, whereas COBPAKM leaves the total number of words occupied by the packed record as an output parameter.

Other differences between the subroutines which are a consequence of this major difference are given in the detailed specifications of the subroutines.

Calling the subroutines

To call one of the subroutines the user writes:

```
                  { COBDIST  }
ENTER PLAN        { COBDISTM } USING identifier-1, identifier-2
                  { COBPAK   }
                  { COBPAKM  }

{identifier-3} , {identifier-4} , identifier-5, [identifier-6]
{literal-1   }   {literal-2   }
```

The meaning of the parameters is as follows. Note that any data-name used as a parameter may be subscripted.

1 Identifier-1. This parameter is the name of the source area which should begin and end on a character boundary.

2 Identifier-2. This parameter is the name of the receiving area which should begin and end on a character boundary and must not overlap with the source area.

3 Identifier-3 or literal-1. This parameter gives the size of the receiving area, in characters for COBPAK, COBDIST and COBDISTM and in words for COBPAKM. It must be either the name of an unsigned DISPLAY field four characters in length (identifier-3) or a four character numeric literal (literal-1), for example 0009.

4 Identifier-4 or literal-2. This parameter provides information about the key specified by the user and in the form described below. It can be either the name of a character field (identifier-4) which contains the key information or an alphanumeric literal (literal-2) which is itself the key information.

5 Identifier-5. This is the name of a four character field used by the subroutines as a reply-word for information about errors in the data or keys as described on page 164. The field must begin on a word boundary.

6 Identifier-6 This parameter is the name of a four character field described as COMPUTATIONAL or COMPUTATIONAL-1 and SYNCHRONIZED RIGHT (identifier-6). Its use is as follows for each of the subroutines:

(a) COBDIST does not use this parameter.

(b) COBDISTM uses it as an input parameter giving the number of words in the source area.

(c) COBPAK uses it as an output parameter giving the number of characters, including field-terminators and the newline characters in the output area.

(d) COBPAKM uses it as an output parameter giving the number of *words* in the output area. Since magnetic tape output has no block terminators any spare characters in the last word of the output area are zero-filled by the subroutine.

In the case of the input or output area not starting on a word boundary the word count starts from the word boundary immediately preceding the start of the input or output area. Similarly, the last word containing any data even if not completely filled is counted as a whole word.

Key formats

The keys which constitute the fourth parameter are a means of indicating to the subroutine the type and length of the variable and fixed length fields of a record. They consist of a string of characters whose meaning is described below and are terminated by an asterisk (*).

At the beginning of the string of characters is *the item or field-terminator* which the user is using to indicate the end of each item or field of input or output data. It consists of any single character or pair of characters except upwards arrow (↑) and the newline character (upwards arrow, asterisk ↑ *) and is followed by an oblique stroke (/) which separates it from the other characters of the key.

The item-terminator is omitted by the unpacking subroutines on input and inserted by the packing subroutines on output.

Both unpacking subroutines will accept input records in which the number of item-terminators in the data is less than the number implied by the key-description. In this case they will assume that the missing item-terminators represent a series of zero or blank fields at the end of the record.

Conversely, both packing subroutines will omit item-terminators representing zero or blank fields at the end of a block.

COBDIST, but not COBDISTM, will accept data even if the last non-blank field of a record does not have an item-terminator, since it is always terminated by a newline character. COBPAK and COBPAKM, however, will always insert an item-terminator after the last non-blank field of a record.

The following items describe each of the fields in turn and begin with one of the following characters: A, F, N, I, D, R whose meanings are as follows. (The lower case characters n, m, j and k represent decimal characters.)

A nnn... indicates that the next field is alphanumeric and contains nnn... characters.

The unpacking subroutines will move data character by character from the next position of the string of input data to the next position of the receiving area until an item-terminator is met. If the number of characters transferred is less than nnn... and consequently the receiving area has not been filled, the subroutines will space-fill the remaining character positions of the field.

The packing subroutines will transfer up to nnn... characters to the output area, inserting an item-terminator after nnn... characters or after the last non-space character on the right if less than nnn... characters have been transferred.

F nnn... indicates that the next field is a fixed length field and does not have an item-terminator.

Both packing and unpacking subroutines will move nnn... characters from the input string to the output string.

N nn mm indicates that the next field is numeric with nn decimal places before the decimal point and mm places after it.

The packing subroutines deal with fields of this type as follows:

1 If there is no leading sign character: (For the purposes of these subroutines only + and - count as sign characters, i.e. overpunched signs, spaces and quote as in DISPLAY, DISPLAY-1, DISPLAY-2 fields are not counted as sign characters.) The subroutines move across up to nn characters omitting all leading zeros or spaces; a decimal point, i.e. an explicit '.', is then inserted into the output areas; up to mm characters are then moved across after the decimal point, omitting all trailing zeros and, finally, the item terminator is inserted.

2 The action for signed fields (DISPLAY-3, DISPLAY-4 and DISPLAY-5 fields) is very similar except that the suppression of leading zeros begins after the sign character. In other words, the sign is floated up to the most significant non-zero or non-space character before the decimal point. It is not floated beyond the decimal point. The count nn includes the sign character.

The subroutine does not check that the characters in a field described by N are numeric. Hence, N can be used to describe fields with an overpunched sign and sterling fields. In the special case of sterling DISPLAY-1 fields, 11d may be represented by a minus sign (-) which the unpacking subroutines might confuse with a leading minus sign. To avoid this, if a sign character occurs in the middle of an input field and if all previous characters are space or zero, not all the leading zeros or spaces are omitted. The one immediately before the sign character is replaced by a space.

EXAMPLES OF PACKING (The item terminator is assumed to be *)

Key	Contents of source field	Contents of receiving area
N 03 03	123456	123.456*
N 03 03	003400	3.4*
N 03 03	▽03400	3.4*
N 03 03	▽▽▽100	.1*

Key	Contents of source field	Contents of receiving area
N 03 03	-▽0001	-.001*
N 03 03	▽▽▽000	*
N 00 03	123	123*
N 00 03	010	01*
N 03 00	-▽1	-1*
N 03 00	+▽▽	*

Notes:

(a) If on packing there are no characters left in the item except possibly a sign and a decimal point, these characters also are omitted.

(b) If the key is N 00 mm or N nn 00 an explicit '.' is not inserted.

The action of the unpacking subroutines is the reverse of that of the packing subroutines. The subroutines examine the first character of the input item to see if it is a + or -. If it is, it is moved to the first character position of the receiving area. The remaining characters up to the decimal point ('.') are space-filled on the left to make them nn characters (or nn - 1 if there is a sign) and moved to the receiving area. The decimal point is omitted and the remaining characters up to the item terminator are moved to the next positions of the receiving area. If the number of characters is less than mm, the remaining character positions are zero-filled.

EXAMPLES (The item terminator is assumed to be *)

Key	Contents of source field	Contents of receiving area
N 03 03	*	▽▽▽000
N 03 03	+1.*	+▽1000
N 03 03	-.001*	-▽▽001
N 03 03	▽-.*	▽▽-000
N 00 03	001*	001

I nn mm kk has the same meaning as N except in the following respects:

(a) On packing, no explicit decimal point is inserted into the output string. In addition, exactly kk characters of the data before the implied decimal point must be moved to the output string even if some of the kk characters are leading zeros or spaces. kk must not be greater than nn and kk is normally equal to nn.

EXAMPLES (The item-terminator is assumed to be *)

Key	Contents of source field	Contents of receiving area
I 03 03 02	▽22200	222*
I 03 03 02	▽▽▽010	▽▽01*
I 03 03 02	-23000	23*
I 03 03 03	-23000	-23*

(b) On unpacking, the action is identical to that taken for an N nn mm key except that instead of searching for an explicit decimal point, the implied point is assumed to be kk characters from the beginning of the item (i.e. the integral part of the item is considered to be fixed-length, kk characters long).

EXAMPLES (The item-terminator is assumed to be *)

Key	Contents of source field	Contents of receiving area
I 03 03 03	+▽2*	+▽2000
I 03 03 03	▽01*	▽01000
I 03 03 02	▽-23*	▽▽-230

D nn mm kk has the same meaning as I except that in this case the fractional part of the variable length item is considered to be fixed-length, kk characters long. The integral part is treated as for N. kk is normally equal to mm.

EXAMPLES (The item-terminator is assumed to be *)

Key	Contents of source field	Contents of receiving area
(a) Packing		
D 03 02 02	-▽010	-10*
D 03 02 02	+▽;00	+;00*
(b) Unpacking		
D 04 03 02	R00000*	R000000
D 04 03 02	+01*	+▽▽▽010
D 04 03 02	▽-111*	▽▽ -1110

R nn mm indicates that the following nn keys are repeated mm times. For example, A4F5A4F5A4F5 is equivalent to R0203A4F5. Note that R may not control R.

Error reports

As mentioned on page 161, identifier-5 in the parameter list is the word into which the subroutines put information about errors detected in the data or the keys. It must have the following description or some equivalent in the Data Division.

01 A

02 B PICTURE 9

02 C PICTURE 1(18)

If on exit from the subroutine no error has been detected, A will be zero. If an error has been detected, however, B will have one of the following values:

1 means that the receiving area has been filled before all the items in the source area have been packed or unpacked.

2 means that the key information is invalid.

3 means that on unpacking a packed data item contains more characters than allowed for by the corresponding key.

For example:

Key	Item
N 02 02	.323*

This error occurs only with COBDIST and COBDISTM. It may also be flagged if the packed data item on D and I keys contains fewer than kk characters

For example:

Key	Item
D 02 04 04	123*

4 means that on unpacking the end of the keys has been reached before the end of the packed source area.

For example:

Key	Source area
↑U/A1A2*	Z↑UBB↑UA↑*

This error occurs only with COBDIST and COBDISTM.

5 means that the receiving area has not been filled. This error occurs only with COBDIST and COBDISTM.

6 means that identifier-3, which must be numeric, contains an alphabetic character.

If the error is type 4 or 6, field C will be zero on exit from the subroutine. If the error is type 5, field C will contain the number of unfilled characters in the receiving field. If, however, the error is any other type, field C will contain the number of the key within the key area on which the error has been detected.

Example programs using the subroutines

COBPAK

Suppose an output area described as

01 OUTPT SIZE 120.

This area is to contain a string of variable-length items each terminated by the characters †U. The items are to be packed after processing from a Working-Storage record with the following description:

01 WORK-STORE.

02 NUM1 PICTURE 9(4)V9(3).

02 NUM2 PICTURE 9(7)V9(2) DISPLAY-1.

02 ALPH1 PICTURE X(4).

02 COM-1 PICTURE 1(48) COMP-1 SYNC RIGHT.

02 NUM3 PICTURE S9(4).DISPLAY-4.

02 NUM4 PICTURE S9(4).

The user could pack this sequence of fields into the area OUTPT by writing the procedural statement:

ENTER PLAN COBPAK USING WORK-STORE OUTPT 0120

"†U/N0403D070202A4F8N0500N0400*" ERROR-REPLY PACKED-CHARS.

where ERROR-REPLY and PACKED-CHARS conform to the specifications for identifier-5 and identifier-6.

If the contents of the source fields were as follows:

NUM 1 ▽▽99990

NUM 2 001311&50 (i.e. £13/11/10.5)

ALPH1 AND▽

COM1 #0000000 00 05 00 44

NUM3 - 000 - (i.e. - £0.0.11d).

NUM4 J123 (i.e. - 1123)

The contents of OUTPT on exit from COBPAK would be

99.99†U 1311 &.50†U AND†U 0000050D - - †UJ123† U†*

The contents of PACKED-CHARS would be 42 and of ERROR-REPLY zero. The remaining characters of OUTPT would be unchanged by the subroutine.

Note that is this case the size of the 'packed' output area is greater than that of the 'unpacked' input area; this is because only a small amount of suppression was possible.

The set of keys given in the parameter-list is by no means unique. NUM2, for example, could have been described as N0702 or I070207, in which case its value in the output area would have been 1311&.5† U or 001311&5† U respectively. It is usually better to use the N key rather than the I or D keys whenever possible since more complete suppression is obtained, but their use may be necessary in some cases for compatibility with other programs.

The field COM1 is of particular interest in this example since it is not a character field, whereas the subroutine deals only with characters. However, as is shown, any block of characters containing non-character fields can be dealt with by writing an F key, in which case nnn... is the total number of characters in the block.

Since the COBOL WRITE statement punches a fixed number of characters when applied to a record to be punched on paper tape, difficulty can be encountered when punching variable length records. This is

because the character positions of the punch record area not filled by a record shorter than the record area are filled with unpredictable characters. To avoid this the user can fill his punch record area with run-out characters before moving the data into the area, but this is inconvenient as well as slow since run-out is a double character in store. A more satisfactory solution of the problem is to use the subroutine PUNCHN which will cause the exact number of characters in a record to be punched. It is called by the statement.

ENTER PLAN PUNCHN USING record-name field-name

in which the parameters have the following meaning:

Record-name is the name of a record of a file to be punched on paper tape.

Field-name is a single word integral field which is described as COMPUTATIONAL or COMPUTATIONAL-1 and SYNCHRONIZED RIGHT and whose value is the number of characters the user wishes to be punched.

The user should therefore follow the ENTER PLAN COBPAK statement in the example with the statement

ENTER PLAN PUNCHN USING OUTPT PACKED-CHARS

The following points should be noted when using the subroutine PUNCHN:

1 In general, COMPUTATIONAL and COMPUTATIONAL-1 fields should not be presented as input to PUNCHN, since it punches in 'normal mode'. In general, the contents of the data fields in the record-area should be restricted to the ALPHANUMERIC character set (see Appendix 1) excluding $] ↑ ←.

2 The file to be punched must be OPENed and CLOSEd by COBOL statements.

3 PUNCHN will not punch more than 128 characters, the maximum size permissible for a record on paper tape. If requested to punch more than 128 characters, the subroutine will stop and display the message:

PUNCHN:- BLOCK GREATER THAN 128 CHARS

If restarted, it will punch only the first 128 characters and then exit.

If requested to punch more characters than the number of characters allocated to the file-buffer the subroutine will stop and display the message:

PUNCHN:- BLOCK GREATER THAN BUFFER SIZE

If restarted it will punch only the number of characters in the buffer.

COBDIST

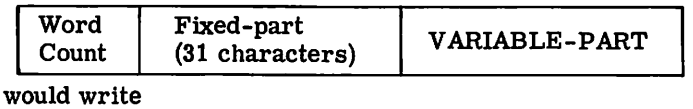
The above example for COBPAK can be reversed to give an example of unpacking variable length fields using COBDIST. If OUTPT is regarded as the source area and WORK-STORE as the receiving area the ENTER function and its parameters are identical except that parameter 3 (0120) is replaced by 0037 and parameter-6 is omitted. Assuming OUTPT contains the same values as before, the contents of WORK-STORE after calling COBDIST will be the same as the contents assumed for the COBPAK example except that the items described with an N key will have leading spaces wherever leading zeros were given in the previous example.

COBDISTM and COBPAKM

Detailed examples are unnecessary since the method of packing and unpacking is the same as for COBDIST and COBPAK with the exceptions mentioned. However, it will be useful to show how the subroutines fit in with the magnetic tape READ and WRITE instructions.

For example, suppose the user wished to read a magnetic tape record with variable length fields and distribute it entirely into a Working-Storage record.

If the input record had the following form:



the user would write

ENTER PLAN COBDISTM USING WORD-COUNT WORK-STORE-RECORD (length of destination) "(item-terminator)/F35 (keys for variable part)" ERROR-REPLY WORD-COUNT.

WORD-COUNT is used twice, as the address of the beginning of the source area and as the count of the total number of words. If the user wished to distribute only the variable part he would replace the first WORD-COUNT with VARIABLE-PART and precede the ENTER statement with the statement:

```
SUBTRACT 8 FROM WORD-COUNT.
```

Similarly, if the fixed part of the record was already in the output record area and the user wished to condense WORK-STORE-RECORD into a variable length string in VARIABLE-PART and then write the complete record, he would write:

```
ENTER PLAN COBPAKM USING WORK-STORE-RECORD VARIABLE-PART
(maximum length of VARIABLE-PART) (keys) ERROR-REPLY WORD-COUNT.
ADD 8 TO WORD-COUNT
WRITE MT-RECORD
```

Chapter 14 Rapidwrite

INTRODUCTION

The purpose of Rapidwrite

I.C.T. Rapidwrite is a commercial autocode based on COBOL applicable to several computers. This chapter defines the language of Rapidwrite as applicable to the 1900 Series of computers and describes the method of completing the forms and procedure cards used in constructing a source program for the Rapidwrite compiler.

One of the advantages of COBOL as a reference language, the fact that it is in English, is a disadvantage to the programmer who wishes to describe his problem concisely. I.C.T. Rapidwrite is designed to overcome this problem without detracting from other desirable features of COBOL. It does this by offering a unique block building technique utilizing dual purpose procedure cards and preprinted data and environment forms.

The advantages of I.C.T. Rapidwrite (i.e. there are fewer rules to remember and there is less writing and less chance of error), derive from the following aspects: much of the program is preprinted so that the writer is guided to the requirements for completion. The meanings of words are established by position instead of by context with other words. There are very few reserved words to remember. The unit card construction of the procedure enables the program to be built up on the lines of a flowchart with moveable blocks.

Using Rapidwrite

A program is produced by first entering information on preprinted forms and dual purpose cards. This information is then punched into cards to form the Rapidwrite (or source) program. The program is then ready for the compiler. At an early stage, the compiler prints out the program in COBOL form and thereafter treats it as a COBOL program.

As with COBOL, a Rapidwrite source program consists of four divisions, namely the Identification, Environment, Data and Procedure Divisions. Since Rapidwrite is a subset of COBOL reference should therefore be made to the appropriate chapters of this manual for a full description of the nature and characteristics of each of the divisions. This chapter assumes a working knowledge of COBOL and explanations of the various clauses and statements are therefore given by reference to the appropriate COBOL clauses and statements.

Facsimiles of the Rapidwrite forms and cards are given in Appendix 3 and should be used for reference when reading this chapter.

Input

The source program is usually input initially on 80-column cards punched from the completed Rapidwrite forms and cards. It can also be punched on paper tape either as a card image (i.e. fixed length blocks of 80 characters) or in variable length blocks, each block ending with the last significant character on the Rapidwrite Card.

The cards or paper tape can be punched with sequence numbers, in which case all the cards of the Procedure Division must be punched with a sequence number in columns 8 to 13 and all cards of other divisions in columns 6 to 11. Each card should have its division letter in column 5.(Environment = E, Data = D, Procedure = P). The rules for sequence numbers in a Rapidwrite program are the same as for a COBOL program, for which see page 17.

Output

The object program compiled from a Rapidwrite source program is produced in standard program library tape format on magnetic tape.

IDENTIFICATION DIVISION

There is no separate form for the Identification Division in Rapidwrite. The program name is written at the head of the Environment Division form.

In the first statement of the program, the letter E must appear in position 5, the six-character program name (see page 19) must appear in positions 15 to 20 and an eight-character alphanumeric account number may appear in positions 28 to 35.

The information in the boxes "program name", "programmer" and "date" at the top of the Environment Division Sheet appears in the COBOL printout of the source program.

ENVIRONMENT DIVISION

For a detailed explanation of the various statements and clauses of the Environment Division see chapter 4. The way in which they are specified in a Rapidwrite source program is as follows. A facsimile of a Rapidwrite Environment Division form is given in Appendix 3.

Configuration section

SOURCE COMPUTER AND OBJECT COMPUTER

1 Positions 23 to 26 of the SOurce and OBject computer statements must contain 190, where n identifies the particular processor in the 1900 series.

2 The number of words of storage allowed for the object program must appear in positions 34 to 38 of the OBject computer statement. (See MEMORY SIZE on page 23.)

3 The OBject computer statement may also specify:

(a) the segment limit for the object program in positions 42 and 43. (See SEGMENT-LIMIT on page 23.)

(b) the equivalent of the ASSIGN OBJECT-PROGRAM clause in position 62. (See ASSIGN OBJECT-PROGRAM on page 23.) C indicates card reader and P indicates paper tape reader. A space in position 62 has the same effect as omitting the ASSIGN OBJECT-PROGRAM clause.

Input-output section

FILE-CONTROL

1 The Rapidwrite equivalent of the COBOL SELECT..... ASSIGN statement (see page 26) is specified by writing AS in positions 13 and 14 and the file name in position 26 of the ASsign statement.

2 The unique one-character file names may be any single letter from A to T.

3 Position 44 contains an I or an O which specifies the file as being input or output respectively.

4 The hardware names +Cn, +Pn, +Mn and +Wn, which specify the input/output unit allocated to each file, must appear in positions 36 to 38 of the ASsign statement.

C indicates a card reader if position 44 contains I, or a card punch if position 44 contains O.

P indicates a paper-tape reader if position 44 contains I, or a paper-tape punch if position 44 contains O.

M indicates a magnetic tape deck.

W indicates a line printer.

n in position 38 must be 1 for the first unit of each type, 2 for the second and so on.

I-O-CONTROL

A DUmp statement is used to specify rerun procedures.

An * must appear in position 26 and the magnetic tape file name must appear in position 50 of the statement. (See page 32 for the COBOL equivalent of a DUmp statement.)

DATA DIVISION

For a detailed explanation of the various statements and clauses see chapter 5. The way in which they are specified in a Rapidwrite source program is as follows. A facsimile of a Rapidwrite Data Division form is given in Appendix 3.

All statements in the division must have a D in position 5. File name W is used to denote the Working-Storage Section (see page 39). (Constant and Linkage Sections do not occur in 1900 Rapidwrite.)

File description

1 The level number FD (see pages 36 and 42) must be written in positions 13 and 14.

2 The unique file-name, one letter in the range A to T, must be written in position 15.

3 The number of characters or records in the block must be written in positions 21 to 24 right-justified Position 25 must contain C indicating characters or R indicating records. A space is taken as indicating characters. (See BLOCK CONTAINS on page 37.)

4 For magnetic tape files, one of the following letters must be written in column 38:

S meaning STANDARD, or

G meaning STANDARD WITH GENERATION-NO.

For all other files an O must be written in column 38. (See LABEL RECORDS on page 37.)

5 File identification (see VALUE of ID on page 19), can consist of up to 12 numeric characters in columns 46 to 57 *or* up to 12 alphanumeric characters provided that quotes are written in column 45 and after the last character.

6 The active time (see ACTIVE-TIME on page 38) must not exceed three numeric digits and must appear in positions 60 to 62 right-justified.

7 Each type of record contained within a file may be listed in a statement equivalent to the COBOL DATA RECORDS statement (see page 38). Each statement must have the level number DR and all such statements must follow immediately after the file description statement. DR statements may be omitted without affecting compilation, provided there are no other errors.

Record description

1 Data names must be unique, consisting of one to five characters and must appear in positions 15 to 19 of the statement. The data name FILLR (see FILLER on page 46) should be used for all fields that are not referenced in the Procedure Division.

2 Only level numbers in the range 01 to 10 and 77 are permitted. They must appear in positions 13 and 14. (See page 42 for level numbers).

3 The number of times that an item occurs (see OCCURS on page 48) must be written in positions 75 to 79. This must be a right-justified positive integer.

4 Every elementary field must be described by a Picture. (See PICTURE on page 49.) It must appear in positions 21 to 43 left-justified. It may contain up to 30 of the characters listed on pages 18 to 24, but if it exceeds 23 characters, it must be continued in positions 21 to 27 on an extension statement which must contain a D in position 5 and an E in position 12. (It may also contain an extension of a VALUE statement (see page 172).

5 The effect of a COBOL REDEFINES statement (see page 54) is obtained by writing an R in position 68 and the name of the item being redefined in positions 69 to 73 of the redefining field.

The fields must be same number of bits in size.

6 Position 44 is used to specify synchronized fields (see SYNCHRONIZED on page 58).

R is equivalent to SYNCHRONIZED RIGHT

L is equivalent to SYNCHRONIZED LEFT

7 The effect of the COBOL USAGE clause (see page 59) is obtained by writing one of the following single characters in position 20:

D indicating DISPLAY

S indicating DISPLAY-1

2 indicating DISPLAY-2

3 indicating DISPLAY-3

4 indicating DISPLAY-4

5 indicating DISPLAY-5

C indicating COMPUTATIONAL

B indicating COMPUTATIONAL-1

If position 20 is left blank DISPLAY is assumed.

8 The effect of a COBOL VALUE clause (see page 62) is obtained by writing the literal in positions 45 to 67. If the literal is alphabetic or alphanumeric, quotes must appear both in position 45 of the first card on which the literal appears and after the last character of the literal. If the literal exceeds 23 characters, it must be continued in positions 45 to 67 on one (or more) extension statements which must have a D in position 5 and an E in position 12. The maximum length of a literal in 120 characters. The extension statement(s) must not contain any further information except a possible extension of a Picture (see page 171).

THE PROCEDURE DIVISION

As in COBOL the Procedure Division is composed of one or more sections, each of which is composed of one or more paragraphs. Paragraphs are composed of sentences and statements. For a complete discussion of the structure and elements of the Procedure Division see pages 65 to 68.

The following are the types of statements available in Rapidwrite. They are identified by the COBOL wording used in the printed version of the program produced by the compiler.

Type of statement	Type of card used
ACCEPT	MOVE
ALTER	ALTER
CLOSE	CLOSE
COMPUTE { ADD / SUBTRACT / MULTIPLY / DIVIDE }	COMPUTE
DISPLAY	MOVE
ENTER	INCLUDE
GO	GO
IF	IF
MOVE	MOVE
NOTE	NOTE
OPEN	OPEN
PERFORM	PERFORM
READ	READ
STOP	STOP
WRITE	WRITE

In addition to the cards mentioned above, a PARAGRAPH card is used to define each paragraph, and a SUBSCRIPT card for indexing table operands.

As can be seen from the above list there is almost a one-for-one correspondence between the available statements and cards. More than one card is used to represent a statement either because subscripts (see pages 66 and 179) are being used, or because one or more cards are used as an extension of the first.

Linking cards

The manner in which cards are grouped to form sentences is indicated by the information punched in column 14 and written by the programmer in box 2 of the card. Box 2 may contain the following values.

1 a blank space, meaning that the card on which it occurs has no logical connection with the card preceding it,

2 an E, meaning that the card on which it occurs continues the statement started on a previous card or cards,

3 a T, meaning that the card on which it occurs represents the first statement depending upon the truth of the condition defined on the card or cards immediately preceding it,

4 an A, meaning that the card on which it occurs represents a statement depending upon a previously defined condition. The performance of the statement depends upon the same condition on the most recent card preceding it containing T. The statement on the A card is performed if the preceding card contains a T and the condition is true.

In the printed version of the program T appears as THEN and A appears as AND.

Sentences which may consist of one or more cards are grouped into paragraphs and sections by preceding them with a PARAGRAPH card (see page 177).

Rules for writing on cards

Information is written on the procedure cards and, using these cards, punched cards are prepared. Printed boxes on the card serve as guides to the programmer when writing. All cards have the following format, although in some cases there are boxes which, or parts of which, are not needed. This is indicated on the printed card by shading printed in the box.

Columns	Box	Use
1 to 4	-	Program number (see page 170).
5		Procedure letter. The compiler checks that all cards have a P punched in this column.
6 and 7	-	Card Code. Punched with one of the different codes to indicate the type of procedure card. The code is always the first two characters of the identifying name printed on the card.
8 to 13	1	Sequence number.
14	2	Either T (THEN) or A (AND) following an IF card, or E for extension.
15 to 20	3	Used for different purposes according to the type of card. The information may be:
21 to 26	4	the name of the paragraph,
27 to 32	5	the number of a section.
33 to 38	6	the name of a file,
39 to 44	7	the name of a record or field,
45 to 50	8	the name of a condition,
51 to 56	9	a numeric literal,
57 to 62	10	an alphanumeric literal,
63 to 68	11	a figurative constant,
69 to 74	12	one or more arithmetic operators,
75 to 80	13	one or more logical operators,
		brackets, or
		an abbreviation for a facility
		available with a particular card.

There are generally 6 positions in each box which are numbered 1 to 6 from left to right. A name is written starting at position 1 of a box, i.e. left-justified. A paragraph name is up to five characters long.

A numeric literal is written so that its least significant digit is in the sixth position of the box, i.e. right-justified. The most significant digit may be + or - to indicate positive or negative. A decimal point (written .) may be included to indicate the assumed position of the decimal point in a MOVE or arithmetic operation. The maximum size of a numeric literal is 6 digits including the sign and decimal point if present. Sterling literals which contain 10d or 11d are not permitted in procedure cards.

An alphanumeric literal is written starting in position 2 of the box, counting from the left. It is preceded in position 1 by a quotation sign and is followed immediately by a quotation sign. The maximum length of an alphanumeric literal, excluding the quotation signs, is 4 characters.

The figurative constant +ALLx may be used, where x is any single character from the alphanumeric set. It produces a string of characters identical to the x chosen, the number of them being deduced by the compiler from context. The constant +ALLx must be left-justified.

Arithmetic operators must appear only on COMPUTE cards and logical operators only on IF cards. Arithmetic operators are always written in position 1 of the box concerned and logical operators must always commence in position 1.

Arithmetic operators	Meaning
+	plus
-	minus
*	multiplied by
/	divided by
Logical operators	**Meaning**
>	is greater than
<	is less than
=	equals
NOT>	is not greater than
NOT<	is not less than
NOT=	is not equal to

Procedures

In the pages that follow the various procedural facilities available in 1900 Rapidwrite are explained in alphabetic order by reference to their COBOL equivalents.

ACCEPT

COBOL Equivalent: ACCEPT identifier

Verb Code: MO

Notes:

1 A MOVE card is used to record an ACCEPT.

2 The hardware name +K1, indicating the console typewriter, must be written in box 4 and the name of the destination field must be written in box 6.

3 For ACCEPT see page 72.

ADD

See COMPUTE on page 175.

ALTER

COBOL Equivalent: ALTER procedure-name-1 TO PROCEED TO procedure-name-2
[procedure-name-3 TO PROCEED TO procedure-name-4 ...]

Verb Code: AL

Notes:

1 Procedure-name-1 must be written in box 3 and procedure-name-2 in box 4.

2 A separate card is required for each subsequent pair of procedure names.

3 For ALTER see page 75.

CLOSE

COBOL Equivalent: CLOSE file-name-1 [file-name-2 ...]

Verb Code: CL

Notes:

1 File names must be written in position 6 of boxes 3 to 13.

2 N in position 5 of any box, specifying CLOSE WITH NO REWIND, is accepted but does not affect the object program.

3 For CLOSE see page 77.

COMPUTE

COBOL Equivalent: ADD, SUBTRACT, MULTIPLY, DIVIDE.

Verb Code: CO

Notes:

1 The COMPUTE verb is used for all arithmetic operations in Rapidwrite.

2 The arithmetic operator must be written in the first (leftmost) position of box 6. The four operators are:

+ ADD

- SUBTRACT

* MULTIPLY

/ DIVIDE

3 The data names or literals must appear in boxes 5 and 7.

4 An optional result field for ADD and SUBTRACT may appear in box 3.

5 The compulsory result field for MULTIPLY and DIVIDE must appear in box 3.

6 If box 3 is left blank, the result is stored in the field named in box 5. If it is, box 5 must not contain a literal.

7 If required, rounding of the result may be specified by writing R in the first position of box 4.

8 A SIZE ERROR condition may be catered for by placing in box 13 the name of the paragraph to which control is to be transferred when the overflow is sensed.

DISPLAY

COBOL Equivalent: DISPLAY $\left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}$

Verb Code: MO

Notes:

1 A MOVE card is used to record a DISPLAY.

2 The hardware name +K1, which indicates the console typewriter, must be written in box 6 and the identifier or literal must be written in box 4.

3 To DISPLAY several items, all items must be MOVEd to a group in the Working-Storage Section, which may then be DISPLAYed. All items in such a group must be in character form, i.e. in DISPLAY form.

4 For DISPLAY see page 80.

DIVIDE

See COMPUTE on page 175.

ENTER

COBOL Equivalent: ENTER language-name subroutine-name [USING P1 [P2 ... Pn]].

Verb Code: IN

Notes:

1 The subroutine name must be written in boxes 3 and 4.

2 Position 6 of box 5 must contain 0.

3 The parameters must appear in boxes 6 to 13. If required, extension statements, each with an E in box 2 and further parameters in boxes 6 to 13, may be used.

4 The INCLUDE card is used for ENTER statements.

5 For ENTER see page 83.

EXIT

COBOL Equivalent: EXIT

Notes:

1 There is no explicit EXIT in Rapidwrite but the same result can be achieved by inserting a paragraph that contains no operations and then ensuring that control is transferred to this paragraph for the various end conditions.

2 This paragraph name must be written in box 4 of the PERFORM card.

3 For EXIT see page 87.

GO

COBOL Equivalent: GO TO procedure-name.

Verb Code: GO

Notes:

1 The name of the paragraph or section to which control is to be transferred must be written in box 4.

2 For GO TO see page 88.

IF

COBOL Equivalent: IF { identifier-1 / literal-1 } { [NOT] GREATER / [NOT] LESS / [NOT] EQUAL TO } { identifier-2 / literal-2 } Any imperative statement

Verb Code: IF

Notes:

1 Identifier-1 or literal-1 must be written in box 3.

2 Identifier-2 or literal-2 must be written in box 5.

3 The operator must be written in box 4, starting at position 1. The permissible operators are:

>	printed during compilation as GREATER
<	printed during compilation as LESS
=	printed during compilation as EQUAL TO
NOT>	printed during compilation as NOT GREATER

NOT< printed during compilation as NOT LESS

NOT= printed during compilation as NOT EQUAL TO

4 The imperative statements are written on appropriate cards following the IF, with T in box 2 of the first card and A in box 2 of all other cards.

5 For IF see page 89.

INCLUDE

See ENTER on page 176.

MOVE

COBOL Equivalent: MOVE { identifier / literal } TO identifier-2

Verb Code: MO

Notes:

1 The source field is written in box 4 and the destination field in box 6.

2 For MOVE see page 90.

MULTIPLY

See COMPUTE on page 175.

NOTE

COBOL Equivalent: NOTE character string

Verb Code: NO

Notes:

1 Position 1 of box 3 to position 5 of box 12 inclusive are considered as a continuous area and the contents of this area are printed during the compilation.

2 Extension NOTE cards can be used with E in box 2 and the NOTE continuing in box 3.

3 For NOTE see page 92.

OPEN

COBOL Equivalent: OPEN { INPUT file-name-1 [file-name-2...] / OUTPUT file-name-1 [file-name-2...] } { [OUTPUT file-name-3 [file-name-4...]] / [INPUT file-name-3 [file-name-4...]] }

Verb Code: OP

Notes:

1 Separate statements must be used for input and output files.

2 The file letters are written in position 6 of boxes 4 to 13.

3 For input files I must be written in position 1 of box 3.

For output files O must be written in position 1 of box 3.

4 For OPEN see page 93.

PARAGRAPH

COBOL Equivalent: paragraph-name

Verb Code: PA

Notes:

1 The start of a new paragraph (see page 65) is indicated by inserting a PARAGRAPH NAME card.

The paragraph name should be restricted to 5 characters in length and must be made up from the character set A to Z, 0 to 9 and hyphen. It must contain at least one letter of the alphabet and neither the first nor the last character may be a hyphen.

2 The paragraph name must be written in box 3.

3 For paragraph-name see page 16.

PERFORM

COBOL Equivalent: PERFORM procedure-name-1 [{ THRU / THROUGH } procedure-name-2]

Verb Code: PE

Notes:

1 Procedure-name-1 must be written in box 3.

2 Procedure-name-2 must be written in box 4.

3 For PERFORM see page 94.

READ

COBOL Equivalent: READ file-name AT END imperative statement

Verb Code: RE

Notes:

1 The file letter must be written in position 6 of box 3.

2 The paragraph to which control is to be transferred when the end of file is sensed must be named in box 5.

3 For READ see page 97.

SECTION

COBOL Equivalent: section-name SECTION [priority-number]

Verb Code: PA

Notes:

1 The section name must be written in box 3 followed by the section priority in positions 5 and 6 of box 4.

2 For an explanation of segmentation and priority-numbers see chapter 8.

STOP

COBOL Equivalent: STOP { literal / RUN }

Verb Code: ST

Notes:

1 Box 3 must contain either "RUN" or a literal.

2 For STOP see page 101.

SUBSCRIPTS

COBOL Equivalent: subscripts

Verb Code: SU

Notes:

1 A field is subscripted by using a SUBSCRIPT statement immediately following the statement containing the name of the field. The subscript is written in the box of the same number as that containing the name of the field to be subscripted. More than one field whose names are written on the same card may be subscripted using the same SUBSCRIPT statement.

2 For subscripting see page 66.

SUBTRACT

See COMPUTE on page 175.

WRITE

COBOL Equivalent: WRITE record-name [AFTER { integer / hardware-name / data-name }]

Verb Code: WR

Notes:

1 The record name is written in box 3.

2 Line spacing before printing is obtained by writing A in position 1 of box 5. Box 6 must contain one of following:

(a) an integer, which must be 0, 1 or 2 (right-justified);

(b) a data name;

(c) +WIn, where n specifies the channel on the paper-tape control band, +WI1 is used by convention for 'top of form'.

3 For WRITE see page 103.

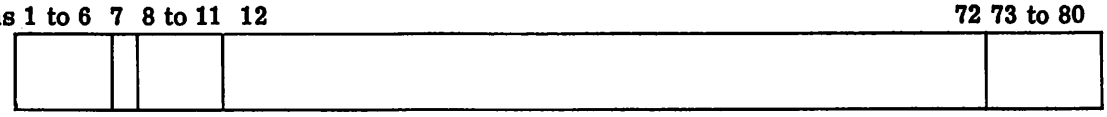
PART 3 OPERATING PROCEDURES

Chapter 15 The Source program

FORMAT OF THE SOURCE PROGRAM

On 80-column cards

The format is:



columns 1 to 6	contain the sequence number, if any,
column 7	contains a space, or a hyphen as a 'continuation indicator',
column 8	onwards may contain division headers, section and paragraph declarations, file descriptions and field descriptions,
columns 12 to 72	may contain procedural statements,
columns 73 to 80	contain the program identification, if any.

For examples of typical COBOL lines in the Data and Procedure Divisions as they would be punched on 80-column cards, see Appendix 5.

An explanation in greater detail of the usage of the various columns follows.

COLUMNS 1 to 6 Sequence numbers may contain spaces instead of zeros in any character positions. The compiler will treat them as zeros and list them as such. The user has the option of punching sequence numbers on all cards, or on none. The compiler takes the following action:

(a) If the first card of the source program has a sequence number, all later cards are checked that they are in ascending order and offenders are flagged but still processed. No sort is performed.

(b) If the first card is unsequenced, the compiler will assign it the number 000100, print the message COMPILER SEQUENCING and thereafter assign numbers to all following cards, incrementing each by 100 from the previous number. These compiler-assigned numbers will be shown on all listings as well as in the corrected source program, if preserved on magnetic tape. If later cards in the source program have sequence numbers, the latter are ignored.

Sequence numbers are compared as 36-bit patterns; if they contain any characters other than numeric, they may have to be changed for compiling on other machines but the first character must always be numeric.

COLUMN 7 This column must contain a hyphen if the last COBOL word on the previous card did not terminate on that card, i.e. it has had to be split over two cards. The following rules for splitting should be noted:

(a) If a non-numeric literal is split over two cards, the 'open quotes' character must be repeated in column 12 or any following column of the continuation card before continuing the literal in the column after that. This applies also in the case where the literal terminates in column 72 of the first card. The second card will then contain, in addition to the hyphen in column 7, an 'open quotes' character in any column from 12 to 71, immediately followed by a 'close quotes' character.

(b) In the case of all words other than non-numeric literals, the first part of the split word need not terminate in column 72 but can be followed by any number of spaces up to and including column 72. The spaces will be ignored and this enables a word to be split on a suitable syllable. The second half of the word must, however, begin in column 12 of the continuation card. This should be contrasted with the case of a non-numeric literal: if it is split and the first card contains spaces in column 72 and preceding columns, the spaces will be considered to be part of the literal.

Examples are given in Appendix 5.

COLUMN 8 ONWARDS All division headers, section and paragraph declarations must start in column 8.

Division headers and section declarations must appear on individual cards; there must be no other punching after the terminal period of the header or declaration, up to and including column 72.

On the other hand, paragraph declarations *may* be followed by the text of the paragraph on the same card, after the terminal full stop and space of the paragraph name.

Field level numbers less than ten can be represented by a single digit, or by two digits, the first being zero. However they are represented on the card, there will always be a zero preceding the single-digit level number in the Data Map listing (see page 197).

COLUMNS 12 to 72 These columns are used in one of three ways:

1 As part of the division header, file or field description etc. that started in column 8.

2 When there is a hyphen in column 7, as the continuation of a COBOL word from the previous card.

3 To contain procedural statements, which begin anywhere after column 11 with any number of spaces, but always at least one, between the separate words of the statement.

Note: COBOL statements and sentences may occupy one or more cards; the hyphen is only needed in column 7 if a *word* is split across two cards.

COLUMNS 73 to 80

These columns may be used for one of two purposes:

1 To hold an identifying name for the program pack: if these columns on the first card are other than blank, the compiler takes their contents and checks that columns 73 to 80 of every card in the pack contain the same identification, and if not, an error is signalled.

2 If these columns on the first card of the COBOL source are blank, the user may write comments or anything else in them, on all or any of the following cards.

LAST CARD The last card of the COBOL source pack, an End Marker, should have four asterisks punched in columns 1 to 4. The contents of the remaining columns are immaterial. The End Marker card should be followed by a blank card.

On paper tape

The format of a COBOL source on paper tape differs from the format on cards in one respect only, the use of horizontal tabs to minimize block size and newline characters to allow the omission from the block of non-significant spaces at the end of a COBOL line.

Each block contains a single COBOL line equivalent to an 80-column card, and terminated by newline. Thus, the maximum number of characters that may be punched for a block is 80 + newline, but since up to three horizontal tabs are permitted, the block will usually contain far fewer characters than this. The three tabs may be punched instead of one or more terminal spaces anywhere in:

Columns 1 to 7

Columns 8 to 11

Columns 12 to 72

Examples: (T = tab, N = newline character)

A paper tape block punched as

TPARAGRAPH-X.TN
or TPARAGRAPH-X.N

is equivalent to a card punched as

Columns 1——7	8——19	20——80
(spaces)	PARAGRAPH-X□	(spaces)

A paper tape block punched as

000100TTMOVE_ITEM_TO_FIELDTN

is equivalent to a card punched as

```
Columns 1——6  7———11  12————————————29  30———80
        000100 (spaces)  MOVE ITEM TO FIELD  (spaces)
```

A paper tape block punched as

000006-¤SPLAY_"END".¤COBOL-12¤

is equivalent to a card punched as

```
Columns 1——6  7  8———11  12————22  23——72  73———80
        000006  -  (spaces)  SPLAY "END"  (spaces)  COBOL-12
```

A paper tape block punched as

¤¤¤¤

or ¤

is equivalent to a blank card.

From the second and the last example it will be seen that tabs may be omitted between the last significant COBOL character and the newline character.

It should also be noted that the characters £,↑,←,], are read into store as *two* characters, and that there can only be a maximum of 40 such characters in any one block.

On magnetic tape

A COBOL source program on magnetic tape is the product of a previous compilation of the same program, when the source has been preserved in this form by using a *CORRECT Steering line (see page 188).

Among the respects in which it differs from the format of a program in card image form on paper tape are:

(a) Each block contains a variable but integral number of COBOL lines.

(b) Only the significant information in the line is included. The terminal spaces at the end of the line are omitted.

(c) Each line is preceded by a four-character word containing in binary the number of words in the line.

There is also a two-word End Marker block in record format, the first word containing 2, the second, four asterisks.

Format of a Rapidwrite source program

The format on cards is given in Chapter 14.

The format on paper tape is a card image omitting spaces after the last significant characters on the card and terminated by a newline character. No tabs are permitted.

PREPARATION OF THE SOURCE PROGRAM

Segmentation

The rules governing segmentation of a COBOL program may be found in Chapter 7. The user is reminded, however, that all overlay sections (i.e. sections whose priority is equal to or greater than the SEGMENT-LIMIT, or greater than 49 if no SEGMENT-LIMIT) must precede the first permanent section in the source program. Furthermore, the priority of each overlay section in the source program must be equal to or greater than that of the preceding section.

Sections that will go into the permanent part of the object program (i.e. which have no priority, or a priority less than the SEGMENT-LIMIT, or less than 50 if no SEGMENT-LIMIT) may appear in any order after the last overlay section.

The entry point to the program is the first statement after the Procedure Division heading, whether this be an overlay section or not.

Check list

In the past, when COBOL compilations might have consumed a lot of expensive machine time, the user was advised to do a great deal of preliminary desk-checking of his COBOL source. In view of the high speed of a 1900 COBOL compilation this is no longer as important as it was. Nevertheless, the user would still be well advised to check out certain features of his program before sending it for compilation, not merely to cut down error messages, but because certain errors may give rise to unpredictable results including a chain reaction of messages. The features to which particular attention should be given are as follows:

HEADERS All programs should be carefully checked for the presence and correct spelling (including hyphens) of the following headers:

IDENTIFICATION DIVISION	INPUT-OUTPUT SECTION
PROGRAM-ID	FILE-CONTROL
ENVIRONMENT DIVISION	DATA DIVISION
CONFIGURATION SECTION	FILE SECTION
SOURCE-COMPUTER	PROCEDURE DIVISION
OBJECT-COMPUTER	

CONTINUATION LINES

If the user wants his program to be listed as a continuous text and so finds it convenient to split words over two lines, he should make sure that he conforms to the rules for splitting given on page 181. He should also verify that hyphens do *not* appear in column 7 when not required.

FULL STOPS The commonest mistake in writing COBOL programs is the omission of full stops where they are mandatory. Full stops *must* appear

(a) after all headers,

(b) after paragraph and section declarations in the Procedure Division,

(c) at the end of each file description (FD level), record description (01 level), field description (02 to 49 levels), 66, 77 and 88 levels in the Data Division.

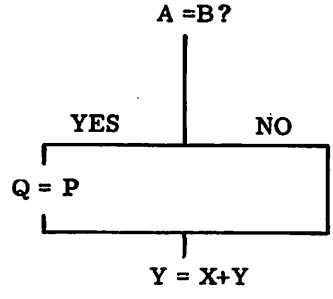
(d) at the end of the program text in each paragraph and section of the Procedure Division.

The following example illustrates the difficulties caused in debugging when a full stop is omitted from a condition.

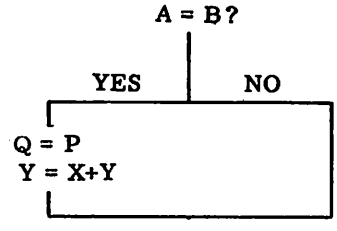
Suppose one wrote

```
IF A EQUAL TO B MOVE P TO Q ⊡
ADD X TO Y ⊡
```

This can be represented by the flowchart:



Now, suppose the full stop was accidentally omitted from the first sentence; the ADD statement would be taken as part of the true exit, and the above would then be represented by:



This logical error in the program might be quite difficult to detect, since if the test program only tests the true exit, the effect will be the same in both cases. Only if it tests the false branch will the error be made apparent, since then 'ADD X TO Y' will not be obeyed.

FILES All files defined in the Data Division must be SELECTed and ASSIGNed in the Environment Division, and vice versa.

All files assigned to magnetic tape or a direct access device must have a one-word field with PICTURE 1(24) defined at the beginning of the record before the user's definition of the record layout to accord with the requirements of the 1900 Housekeeping packages. This field need only be given a name other than FILLER if the user refers to it in his program, e.g. to handle variable length records.

READ Verb The data name must be a file, and all READs must have an AT END branch.

WRITE Verb The data name in a WRITE statement or the first data name in a WRITE...FROM...statement should be a record (01 level).

STOP Verb The statement must include either the word RUN or a literal.

ENTER Verb All literals, whether numeric or non-numeric, supplied as parameters to an ENTERed routine, will go into Upper Storage in the object program.

OPEN Verb All OPEN statements must include the word(s) INPUT and/or OUTPUT.

SEGMENTATION When a source program is segmented, the PROCEDURE DIVISION line must be followed by the first overlay declaration.

LABELS LABEL RECORDS STANDARD must be included in the FD entries for all magnetic tape and direct access files. The clause LABEL RECORDS OMITTED need only be used for all other files for compatibility with other COBOL compilers.

DATA RECORDS This clause need only be used for compatibility with other COBOL compilers. If used, it should contain, in any order, the names of all records (01 levels) defined for the file. No check is made to see whether these match the names given to the corresponding 01 level records.

HYPHENS All data names should be checked for missing hyphens.

PICTURE The symbols V and P are *not* included when determining the size of a data field from a Picture. The symbol S will sometimes imply an extra character or bit of storage, sometimes not, depending on the usage of the field (see pages 59 to 62), but it is never taken into account when determining the maximum *value* that a field can contain.

STEERING LINES

Steering lines are the parameters that tell the compiler what the user wants done. They may be on cards or paper tape. If they are on paper tape, tabs are not allowed and each block must be terminated by a newline character. In both media a line can contain a maximum of 71 characters including spaces but excluding newline in the case of paper tape. Steering lines are the first items of the input to the compiler, and are terminated by an End Marker (card or block) of four asterisks in the first four character positions.

Each Steering line consists of a directive immediately preceded by * punched in the first character position, and sometimes followed by a 'medium' and/or 'operand'. These three items in the line are separated from each other by an optional number of spaces (but at least one). All the information for each Steering line must be on one card or paper tape block, i.e. there are no continuation lines.

The format of lines giving information about the input/output media for the various files used by the compiler (COBOL source program, Correction lines, object program etc.) is as follows:

> Following the directive and separated from it by at least one space is the medium – CARDS, PAPERTAPE, MAGNETICTAPE or NONE. (The last denotes that an optional file is not required or provided and nothing else follows it in the line.) The abbreviation MT may be used instead of MAGNETICTAPE and PT may be used instead of PAPERTAPE, but no other words or abbreviations are permitted as the medium.
>
> Following the medium and separated from it by at least one space is the operand, preceded by an open bracket and terminated by a close bracket. There are two types of operand, both of which must contain a file name which will be used in compilation:

1 *Slow peripheral operands* – those whose medium word is CARDS or PAPERTAPE. Their format is

xxxxxxxxxxxx

where x...x is a file name of one to twelve characters, the first alphabetic and the remainder taken from the set: alphabetic, numeric, space, hyphen.

2 *Fast peripheral operands* – those whose medium word is MAGNETICTAPE. The operand may consist of several units, and the format of a unit is shown on page 187.

A magnetic tape file may also be relabelled. Such a unit in the operand is introduced by an equals sign (=). It should be noted that this is relevant only for output files and that only one such unit is permitted in any one Steering line; also that the optional field containing the tape serial number (aa...aM – see page 187) has no relevance in such a unit.

For example, if the user wished his object program to be written to a scratch tape, the operand of the *OBJECT Steering line might be:

(SCRATCH TAPE = PROGRAM$_{\nabla}$cccc)

where cccc are the first four characters of the name in the PROGRAM-ID clause.

If the user wished the object program to be assigned to a tape already labelled COBOL-TEST14, the operand would be:

(COBOL-TEST14 = PROGRAM$_{\nabla}$cccc)

and if he wished it to be assigned to a tape already labelled PROGRAM CCCC, the operand would simply be:

(PROGRAM$_{\nabla}$cccc)

since no relabelling is necessary.

Note: A Scratch Tape output file must always be relabelled (as above). The 'M' at the end of the Tape Serial Number has no significance at present and may be omitted. Later versions of the compiler will check for the absence of a Write Permit Ring for all files defined, by the presence of the 'M', as 'Masters'.

In addition, a magnetic tape file may have one or more subfiles. For example, a tape may contain a number of COBOL source programs, each being a subfile and all of them together being the file. This type of unit in the operand is introduced by a full stop (⊡).

Units may appear in any combination in the permitted format, and a typical example might be:

(unit A ⊡ unit B = unit C ⊡ unit D)

where unit B and unit D are optional subfiles and could be followed by similar units, unit C is an optional relabelled file, unit A is the only mandatory unit in the line.

The above information on operands does not apply to those after the directives *IDENTITY, *LIST, *DATA and *PROCEDURE, whose operands are not introduced or terminated by brackets.

Use of the Steering lines

There are two types of directive: major and minor. Minor directives, some of which are mandatory, follow and are associated with the preceding major directive and may be in any order within the group of minor directives, but the major directives (not all of which need be used for a compilation) must appear in the order given on pages 188ff.

Steering lines may be used in a variety of combinations for a variety of purposes, as will be seen from their descriptions. The complete list of directives with their functions is given on pages 188ff.

Notes.

1 *IDENTITY and the End Marker (****) must always be present in any selection of Steering lines.

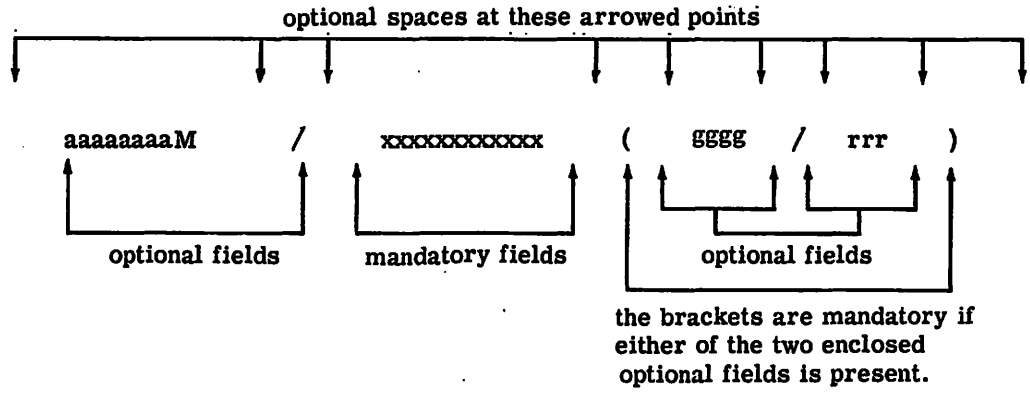
2 If *LIST COBOL appears more than once in a group of Steering lines, the source program will be listed more than once.

The *SUBROUTINES Steering line must not appear more than once.

3 A *COBOL Steering line must always be present if there is a *TRANSLATE and/or *CORRECT and/or *COMPILE Steering line, but whatever the mode of compilation there must never be more than one *COBOL Steering line.

For examples of Steering lines see Appendix 5.

Format of a fast peripheral operand unit



KEY

1	aaaaaaaa	denotes the tape serial number in octal representation, i.e. characters in the range zero to seven – maximum eight digits. (The number is expected to be the same as that which appears on the outside of the tape reel.) The character M, if present, must immediately follow the tape serial number and cannot be used without it; it signifies that the file is a 'master file', i.e. the tape should not bear a Write Permit Ring. The field is terminated by /
2	xxxxxxxxxxxx	denotes the file name of one to twelve characters, the first of which is alphabetic and the rest drawn from the set: alphabetic, numeric, hyphen, space. File names of fewer than twelve characters are space-filled by the compiler. If the file name SCRATCH TAPE is used, the compiler will allocate a scratch tape.
3	gggg	is introduced by (and denotes the file generation number – maximum four decimal digits. The field is terminated by / if the reel sequence number is to follow, otherwise by). If the generation number in a Steering line is zero it will be ignored, not checked by the compiler.
4	rrr	is introduced by / and denotes the reel sequence number – decimal, in the range 0 to 511. The field is terminated by).

Note: The file name, reel sequence number and file generation number form the identification of information on the tape. They provide a means of checking that the correct file is being processed, except that the file generation number is not checked if it is zero, in the Steering line.

List of directives

Major directive	Minor directive	Medium	Operand	Function
*IDENTITY			cccc,nnnnnnnn	cccc is the name of the COBOL source program, being the first four characters of the PROGRAM-ID clause. n...n is the optional ACCOUNT-NO of one to eight decimal digits, space filled if there is no ACCOUNT-NO and introduced by a comma.
*CORRECT				Used when correcting a COBOL source program by inserting, deleting or replacing lines; also used to preserve the source on magnetic tape, or other medium.
	*MASTER	CARDS or PAPERTAPE or MAGNETICTAPE or NONE	As for slow or fast peripherals, or no operand.	Specifies the input medium from which the COBOL source program is to be read.
	*COBOL	CARDS or PAPERTAPE or MAGNETICTAPE or NONE	As for slow or fast peripherals, or no operand.	Specifies the output medium to which the corrected source program is to be written.
	*AMENDMENTS	CARDS or PAPERTAPE or NONE	As for slow peripherals, or no operand	Specifies the input medium from which Correction lines are to be read.

Major directive	Minor directive	Medium	Operand	Function
	*LIST		AMENDMENTS and/or COBOL	Instructs the compiler to print out information according to the operands used. The operands must be separated from each other by one or more spaces. If the user prefers, instead of putting both operands in a single Steering line he can use two *LIST lines.

Note: The *CORRECT Steering line must always be used when it is desired to preserve the source program on magnetic tape. If it is desired to preserve the source program on magnetic tape although making no corrections (as on an initial compilation), the minor directive *AMENDMENTS with NONE is used and no amendments supplied, or *MASTER NONE is used and the source program is quoted as the *AMENDMENTS file. In the latter case, but not in the former, a check is performed on the sequencing of the source program.

The user can if he wishes CORRECT without compiling, that is to say the *CORRECT Steering line need not be followed by a *COMPILE. In this event, the compiler will halt after transcribing the source program to magnetic tape and performing the listings, if any, called for by *LIST, and display the message 'SERVICED #XXXXnnnnnnnn', where XXXX is the program name and nnnnnnnn is the account number.

Major directive	Minor directive	Medium	Operand	Function
*COPY	*SKELETON	CARDS or PAPERTAPE or MAGNETICTAPE	As for slow or fast peripherals.	Specifies input medium from which COBOL source is to be read.
	*COBOL	CARDS or PAPERTAPE or MAGNETICTAPE	As for slow or fast peripherals.	Specifies output medium to which de-COPYed source program is to be written.
	*LIBRARY	CARDS or PAPERTAPE or MAGNETICTAPE	As for slow or fast peripherals.	Identifies file holding the COBOL library.
	*LIST		SKELETON and/or COBOL.	Instructs listing of input and/or output source program.
****				End Marker.

Note: If the *COBOL file is listed, an asterisk is printed in column 82 of all lines copied from the library.

*COMPILE				Orders compilation of a COBOL source program (which may have been translated from Rapidwrite).
	*COBOL	CARDS or PAPERTAPE or MAGNETICTAPE	As for slow or fast peripherals	Must follow *COMPILE unless it has appeared after *TRANSLATE or *CORRECT, in which case it must not be used. See note 3 on page 187.

Major directive	Minor directive	Medium	Operand	Function
	*OBJECT	CARDS or PAPERTAPE or MAGNETICTAPE or NONE	As for slow or fast peripherals, or no operand	Specifies the output medium to which the semi-compiled object program is to be written. The file name must be PROGRAM cccc, where cccc is the four-character name in the PROGRAM-ID clause.
	*LIST		COBOL and/or MAP	Comments as for *LIST above. MAP specifies that a listing of the Program Maps and Data Maps is required.
	*LINKAGE			Specifies that the current COBOL program is a subroutine. It must be present for subroutines.
	*OVERLAY			For master programs only. If present, it specifies that subroutines are to be included in overlays in accordance with standard techniques. If absent all subroutines go into permanent program.
	*STANDARD			Specifies that period in the PICTURE clause implies the decimal point position in a field as well as being an insertion symbol, and that V will not appear with a period in a PICTURE (see also page 139).
*CONSOLIDATE				Orders that a consolidated object program is to be produced from compilation.
	*SUBROUTINES	CARDS or PAPERTAPE or MAGNETICTAPE or NONE	As for slow or fast peripherals, or no operand	Informs the compiler that subroutines in semi-compiled form are to be incorporated in the object program from the specified medium. Must always be used if any user (ENTERed) subroutines or standard compiler generated run time routines are not on the same medium as the compiler itself (e.g. are on a different magnetic tape).
*LOAD				Achieves automatic loading of the object program - see page 251.

Major directive	Minor directive	Medium	Operand	Function
*DATA *PROCEDURE			nnnnn X pp ⊡ q optional spaces	These two lines, if provided, may assist the compiler to optimize the use of core store and may save a pass in compilation if the compiler is being obeyed in more than its minimum store size. nnnnn is a number of one to five decimal digits. After *DATA it denotes the number of data names in the source program: after *PROCEDURE it denotes the number of procedure names in the source program. pp.q is the average number of characters in these names. pp (integral part) maximum 30. q (optional, fractional part) maximum one digit. The multiplication sign (the character X) is obligatory.
*SIZE			nnnnn	This optional Steering Line specifies the core size in words (nnnnn) to which the compiler is to be restricted.

Major directive	Minor directive	Medium	Operand	Function
*LOWER			nnnn	nnnn is a number of one to four decimal digits (for meaning see page 243).
****				End Marker

ORDER OF INPUT

If all the input to the compiler is to be read from the same peripheral (card reader or paper tape reader), the order of the input must be:

Steering lines
End marker (****)

Correction lines
End marker (****) } only if corrections are present

COBOL source program
End marker (****)

COMPILER OPERATING INSTRUCTIONS

General

The actions of the compiling system are controlled by the Steering lines (see page 185).

An example of an operating instruction sheet is given on page 195; this will help to clarify the explanation and is recommended as a model, although users may wish to redesign it to suit their own purposes. References to this sheet in the following text are in italics.

The compiling system is made up of various phases and the programmer should tick those phases which are to take place (the compiler itself must always be loaded and Steering lines must always be present, so the 'SYSTEM LOAD' and 'STEERING' phases are printed with ticks against them). This acts as a check list for the programmer, and notifies the operator what sequence of events to expect.

Loading

The name of the compiler is stated when each new version is issued, and the appropriate name should be inserted where required on the instruction sheet.

The number of work tapes and magnetic tape decks required, and in what circumstances is stated in the document accompanying each compiler, and the appropriate numbers should be entered on the sheet (*two boxes at the head of the page*).

When the system is loaded it should be allocated as much core store as possible (to achieve maximum efficiency) but at least the minimum figure quoted for the compiler on issue. To load the system it is necessary to type a message of the form

FIND #ABCD nnnnn.

where ABCD is the name of the compiler and nnnnn is the amount of core store to be allocated (*enter these in the two boxes after '1.FI '*).

Note that if nnnnn is omitted, compilation will usually HALT displaying the message MT ERR 2.

When a figure nnnnn *is* quoted, then on multiprogramming machines this is the amount of core that will be allocated to the compiler, if available, but on non-multi-programmming machines *all* the available store will be allocated.

The compiler must not be given more than 31,743 words of store in any 1900 processor. The object programs must not request or be allocated more than 32,768 words of store in any 1900 processor.

The final portion of the COBOL Map listing sometimes shows the address of a portion of object programs as 31,743, meaning "not presented to the COBOL Consolidator". Any attempt to obey such a program in an object program of 31,743 or fewer words results in its going illegal. However, in a program occupying between 31,744 and 32,768 words of store instruction 31,743 is obeyed with unpredictable results.

Switch settings

1 The compiler may also be informed of the total number of magnetic tape decks, including the deck on which the compiler itself is loaded, available for the compilation. This is done by setting the switch (in word 30) corresponding to the required number, which must be in the range four to eight inclusive (*enter the number in the box after '2.ON'*). Step 2 of the Operating Instructions may, however, be omitted, that is no switch need be set and in this event the compiler assumes six decks are available.

Note: Since the compiler can operate with a minimum of four decks, users are recommended *either* always to specify 4 in their Operating Instructions but to leave standing instructions with their Machine Room to use up to eight decks if available and useful, *or* if they specify more than four decks in their Operating Instructions, to instruct operators that compilation may nevertheless take place with a minimum of four decks, if available. If this is not done compilations may not be run when the specified number of decks is not available, even though the minimum number is.

2 The compiler's work-files will normally be on the same medium as the system tape. Any contrary requirement will necessitate the following switch settings:

ON 21 (compiler's work-files on magnetic tape)

ON 22 (compiler's work-files on cassette tape)

Hence, if the compiler is operating from magnetic tape and its work-files are also on magnetic tape, neither switch 21 nor 22 should be set. If, however, its work-files are on cassette tape, switch 22 should be set.

Entry points

The entry point to the system depends on the Steering lines medium. Entry 20 should be used when the Steering lines are on paper tape and 21 when they are on cards (*tick one of the boxes against '3.GO'*).

The run is thus started by typing either

GO #ABCD 20, or

GO #ABCD 21

where ABCD is the compiler name.

End messages

All end messages printed by the system on the console typewriter contain the source program and account number (*insert in the boxes next to 'nnnn = OBJECT PROGRAM NAME' and 'aaaaaaaa = ACCOUNT NUMBER'*). The expected end message will start with 'SERVICED' if the last phase to be obeyed is TRANSLATE or CORRECT, with 'COMPILED' if the last phase is COMPILE, or with 'CONSOLIDATED' if the last phase is CONSOLIDATE. (*Tick the appropriate box under Operating Instruction 4.*)

If any errors have been detected during the run which have not brought it to a halt the expected end message will be followed by

pppp ERRORS

where pppp is the number of errors detected.

If, on the other hand, a serious error is detected which halts the run, then the expected end message will not be printed. Instead, in whatever phase the error is detected a message will be printed of the form

SERVICED #aaaannnnnnnn. ABANDON

where aaaa is the source program name, nnnnnnnn is the account number.

The programmer should state on the instruction sheet (*in the boxes under Operating Instruction '5. ERRORS'*) what action is to be taken by the operator if either of these error summary messages is printed on the typewriter.

Special action during consolidation

The SPECIAL ACTION section should only be completed if consolidation is to take place and subroutines, whether ENTERed Own Code or standard routines required by the compiler, are to be input from a slow peripheral. If the compiling system is being obeyed from a Library Tape the restart point is 28, otherwise it is 27 (*enter in the box under SPECIAL ACTION*).

File names

The compiler operating procedure requires that all input and output files should be named in the Steering lines so that these names can be used when communicating with the operator (*insert under 'FILE NAMES' against the System phases and tick the appropriate peripherals*).

If a file is on magnetic tape the Tape Serial Number should be quoted (*T.S.N. column*) and whether the tape is to be mounted with or without a Write Permit Ring (*WPR column*). The Tape Serial Number acts as the physical identification of the file. The file name for the System Tape will be PROGRAM TAPE, if the compiling system is to be obeyed from a Library Tape; otherwise it will be PROGRAM ABCD where ABCD is the name of the compiler. If the System Tape is a Library Tape the Tape Serial Number need not be quoted on the sheet, only the identification, for example, MLT2/10 (*in T.S.N. box*).

Slow peripheral input files originating from the programmer should be marked with the appropriate names (*entered in the 'FILE NAMES' boxes*).

The file name for the Steering lines is always STEERING on the sheet, but on the cards or tape it should be qualified by the program name – STEERING – #ABCD – in order to differentiate between the Steering lines for different programs.

Compiler messages

During the running of the system, messages will be displayed to tell the operator to load and unload peripherals and to label output. Halts will occur only rarely, for example when a peripheral is not available. All messages are self-explanatory but further information is given on pages 201 to 204.

If the console message HALTED MT appears just before or during consolidation, the operator should restart by typing GO.

Should the message HALTED nn be displayed during compilation (where nn is a number in the range 01 to 63), a tape failure has occurred and the program should be recompiled using different tape decks and/or a different Compiler System Tape. The same remarks apply to the displayed message OR (no halt).

This message will also occur, with nn in the range 20 to 29, if the user starts compilation by GOing to a non-standard entry point (i.e. other than 20, 21, 27 or 28).

If the message HALTED JM appears during compilation, it signifies either a hardware error (the program should be recompiled) or the presence of a long and complex IF statement in the COBOL source (the statement should be broken down into a succession of smaller statements, and recompiled).

Messages output by the General Purpose Loader

When a consolidated semi-compiled program containing a General Purpose Loader is loaded as the result of a LOAD or FIND message to Executive, successful termination of loading is indicated by one of the messages:

HALTED LD or

DISPLAY LD

Occasionally the General Purpose Loader may detect an error and will halt with an error message instead of the 'LD' message. The formats of the error halt are:

HALTED – LD ERR-n or

HALTED En

where n is a letter indicating the type of error. The second of these formats will gradually supersede the first. The error halts that may arise are EA, EI, EN, EP, EQ, ER, ES, ET, EX. They can be distinguished from other two-letter messages by the fact that no LD message has yet occurred.

Such error halts should not occur in normal practice. If they do, the usual causes are that the program being loaded has become corrupted or that a hardware failure has led to a misread. The normal restart action is to load the program again from the beginning. If this fails, the program should be re-compiled.

There are two exceptions to this procedure.

1 Error halt EX indicates a checksum failure. If the input is on paper tape or cards it is possible to restart by moving the input back one block or card and inputting a GO message.

2 Error halt ES when reading cards indicates a card sequence error. The cards may be sorted back into correct sequence and an attempt made to continue reading by inputting a GO message.

Messages output by the Overlay Packages

When a program contains overlays the following messages may occur during the running of the object program:

HALTED OVRL ERR 1 HALTED OVRLD ERR 2

or

HALTED E1 HALTED E2

This means that the object program has overwritten certain parts of its data store area, thus affecting the Overlay Package.

Sample operating instruction sheet

SPECIAL INSTRUCTIONS:

1900 COBOL INSTRUCTION SHEET

RUN DESCRIPTION AND PERIPHERAL REQUIREMENTS

MAX. No. OF WORK TAPES [ ]

TOTAL No. OF MAG. TAPE UNITS [ ]

SLOW PERIPHERALS ARE ALLOCATED AS REQUIRED AND RELEASED AS SOON AS POSSIBLE.
A PRINTER IS REQUIRED DURING THE WHOLE RUN.

SYSTEM PHASES		INPUT/OUTPUT FILES	FILE NAMES	SLOW PERIPHERALS				MAGNETIC TAPES	
				CR	CP	TR	TP	T.S.N.	WPR
SYSTEM LOAD	✓	SYSTEM TAPE							
STEERING	✓	STEERING LINES	STEERING						
TRANSLATE		RAPIDWRITE SOURCE							
		TRANSLATED SOURCE							
CORRECT		MASTER SOURCE							
		AMENDMENTS							
		CORRECTED SOURCE							
COMPILE		COBOL SOURCE							
		OBJECT PROGRAM							
CONSOLIDATE		SUBROUTINES							

OPERATING INSTRUCTIONS

(N.B. DISPLAYS AND HALTS DURING THE RUN ARE SELF - EXPLANATORY, BUT FULLER INFORMATION IS GIVEN IN THE 1900 COBOL REFERENCE MANUAL)

1 FI # [ ] [ ] (CORE STORE REQUIRED) FROM SYSTEM TAPE

2 ON [ ] (TOTAL No. OF MAG. TAPES REQUIRED - 6 ASSUMED IN DEFAULT)

3 GO

20		STEERING LINES ON PAPER TAPE
21		STEERING LINES ON CARDS

SPECIAL ACTION DURING CONSOLIDATION

IF SUBROUTINES ARE ON A SLOW PERIPHERAL AND ALL THOSE WHICH HAVE BEEN SUPPLIED HAVE BEEN READ, AND EXECUTIVE TYPES ' UNIT NN FIX,' THEN SUSPEND AND ' GO ' [ ]

4 EXPECTED END-OF-RUN MESSAGE

	SERVICED	#nnnn aaaaaaaa	ACTION:
	COMPILED		
	CONSOLIDATED		
		nnnn - OBJECT PROGRAM NAME	
		aaaaaaaa - ACCOUNT NUMBER	

5 ERRORS

pppp ERRORS	ACTION:
SERVICED # nnnn aaaaaaaa ABANDON	ACTION:

Chapter 16 Compiler listings and other printouts

INITIAL MESSAGES

The first message printed by the compiler is:

(time) (date) 1900 COBOL COMPILER

There follows a listing of the Steering lines, then whatever other listings may have been specified in the Steering lines with interspersed error messages, if any.

These listings are described in detail below.

THE SOURCE PROGRAM

If the Steering line *LIST has been used there will be one or more listings – of the COBOL source, Rapidwrite source, Corrected source etc. – according to the operands in the Steering line.

In all such listings, whether of COBOL or Rapidwrite and whatever the input medium, each line will be printed as a card image beginning at print position 1.

It should be noted that error messages, if any, may be interspersed with the source listing at different places on different occasions, even when the same COBOL program is being compiled, or they may be printed separately from the source listing. This may occur on a multiprogramming machine when the user specifies a different store availability, because, as described in Appendix 6 the units of the compiler program may then be combined in a different number of passes from that of the earlier compilation.

THE DATA MAPS

If MAP is an operand in a *LIST Steering line, the compiler prints out an analysis of the relative run time addresses assigned to each field, and the record area of each file, described in the COBOL Data Division, one line per file or field. These Map listings may be interspersed with the listing of the source program. The format of a line is:

Print position	Printout
47 to 49	MAP
50	blank
51 and 52	FD, or nn (level number)
53	blank
54 to 83	file or field name (space filled at end)
84	blank
85 and 86	UV, UP, LV or LP (for Upper Variable, Upper Preset, Lower Variable or Lower Preset) or blank (for fields in File Section).
87	blank
88 to 92	relative word address, right justified
93	period
94	character address within word
95	period
96	bit address within character
remaining print positions	blank

A number of lines follow, one for each file, with the word FILE-LIST in columns 42 to 50, the name of the file in columns 54 to 83 and the Map lines in columns 84 to 96 as above.

Notes:

1 Print positions 93 onwards will be blank for FD levels.

2 The level number is not zero suppressed.

3 All other numeric fields are zero suppressed except for the last digit.

4 Examples of Data Map listings are given in Appendix 5.

THE PROGRAM MAPS AND SEGMENT LISTING

The listings so far described are printed during the Analysis phase. At the beginning of the Generation phase, the following lines are printed:

	MAP-PROGRAM-NAME-cccc	(where cccc are the first four characters of the name given to the COBOL program in PROGRAM-ID)
and	MAP-SECTION-(name)-PERMANENT	
or	MAP-SECTION-(name)-OVERLAY-xx	(where (name) is the name of the first section in the COBOL program or, if it is not divided into sections, the word COBOL.)

Note: If the source program is divided into sections but the writer has failed to begin the Procedure Division with a section declaration, the name COBOL is given to the first section and the whole of the object program will go into Permanent, i.e. overlaying will be ignored.

xx is the priority number of the section.

The second line is only printed instead of the first if the priority is equal to or greater than the SEGMENT-LIMIT, or greater than 49 if there is no SEGMENT-LIMIT.

Thereafter, at the beginning of generation of final object code for each *segment* (i.e. group of COBOL sections with same priority, whether Permanent or Overlay), except the first, a line is printed with the same format as the PERMANENT or OVERLAY messages above.

At the end of generation of code for each segment a line is printed with the format

MAP-SECTION-(name)_____ { PERMANENT_PROGRAM nnnnn / OVERLAY

where nnnnn is the number of instructions/words generated for the previous overlay or permanent segment.

Between the initial and terminal messages for each segment there will also be a line printed for each COBOL paragraph. The format is

MAP-SEQ-nnnnn-PRIORITY-yy-ADDRESS-ppppp

where nnnnn is the sequence number of the first COBOL line in the next sentence.

yy is the priority number (00 if permanent) of the segment in which the sentence appears, ppppp is the run time address within the program relativizer of the *next* word to be generated.

(Users should note, however, that the current versions of the compiler print such a line without 'PRIORITY-yy' for each COBOL *sentence,* not for each paragraph only.)

The user can thus see at a glance, by subtracting the *ppppp* value of one such message from that of the next, how many words of object code have been generated for the COBOL paragraph beginning in line *nnnnn* (as quoted in the first of the two messages).

Using this calculation and the segment message giving the total number of words generated for the segment, the user is thus enabled to resegment, if necessary, by moving paragraphs from one part of the program to another.

Examples of program maps and segment listing are given in Appendix 5.

Note: As in the case of Data Maps, Program Maps and Segment Listings will only be printed if the operand MAP has been used in a *LIST Steering line.

The consolidator listing

The Consolidator Listing is the last printed output of the compiler if the *CONSOLIDATE Steering line has been used and if there is also an *LIST MAP Steering line, in which case it *follows* the 'Terminal Messages of Compilation' given on page 239.

The present listing is as follows:

first line: CORE nnnnn (always printed, even if there is no *LIST MAP Steering line).

This gives the total amount of storage required by the object program after consolidation, i.e. after all Own Code routines, M.T.H. and compiler-generated routines have been incorporated in the object program.

second line: Relativizers (now restricted to Lower Variable Lower Preset, Literals, Upper Preset and Upper Variable) printed as:

LV nn LP nn LT nn UP nn UV nn

where nn is the start address in the object program of these areas. These addresses may be used in conjunction with the Data Map, Program Map and Segment Listings to debug the object program from core dumps. In versions of the compiler incorporating COBOL Subroutine facilities, these relativizers give the start address of subroutine private (i.e. non-Linkage) data areas. The addresses of the master program's data areas are given by the common area cues %LV, %LP, %UV and %UP.

The cue list follows. This is now restricted to types 41 and 45 cues (these numbers being the first two digits in a cue line), and type 00 – 'unsatisfied cues'. *Type 41* cues are those in the Permanent part of the object program and comprise the names (or entry points) of users' ENTERed routines, M.T.H. and compiler-generated routines. When the first type 41 cue is met the message

MAP PERMANENT PROGRAM BEGINS AT nnnn

is printed, then the cues themselves in the format

41nnnn FRED

where nnnn is the decimal address of the cue in Permanent.

Type 45 cues are compiler-generated, and are only present if the object program is segmented into overlays. The cue lines themselves are not printed, only the single message

MAP OVERLAY PROGRAM BEGINS AT nnnn

where nnnn is the decimal start address of the overlay area in the object program.

ERROR MESSAGES

Two types of error may be distinguished:

1 Format and parameter errors, such as incorrect Steering lines or combinations of Steering lines, source programs in incorrect format etc.

2 COBOL language errors: elements and combinations of elements in the COBOL source program not used in accordance with the rules of COBOL.

Where the message are output on the line printer the format is:

ERROR_____SEQ__nnnnnn__REF__ppp (message)

where ppp is the error number, nnnnnn is the sequence number of the first line of the sentence in which the error occurs, or in the case of a sequence error, the number of the offending line.

A complete list of messages is given on pages 201 to 233, and the following notes should be used in conjunction with the lists to interpret the messages, and the action required.

1 In the case of the messages on pages 201 to 204 the action to be taken by the programmer is given. In the case of messages on pages 204 to 233 only the action taken *by the compiler* is given, since

programmer action will always be necessary at some stage to make the corrections to the source program and recompile, if only to obtain a clean listing.

2 The term 'quasi-character field' is used in explaining certain Generation phase error messages. It means a field which starts on a character boundary and is a multiple of six bits in length (i.e. all DISPLAY-n fields, and certain COMPUTATIONAL and COMPUTATIONAL-1 fields, depending on their size and context in other fields).

3 In many of the Analysis phase messages the compiler action is given as 'statement being processed is ignored'. In this context the meaning of 'statement' is any group of words in the source program that begins with one of the words listed below and ends immediately before the occurrence of another of these words. (This is particularly important in understanding 'ricochetting' errors, that is, errors which give rise to a number of different messages because they have caused the compiler to ignore parts of the context of the offending word.)

⊡	ENTER	READ
ASSIGN	ENVIRONMENT	RECORDING
ACCEPT	FILE	REDEFINES
ADD	FILE-CONTROL	RERUN
ALTER	GO	SEGMENT-LIMIT
BLANK	I-O-CONTROL	SELECT
BLOCK	IF	SOURCE-COMPUTER
CLOSE	INPUT-OUTPUT	STOP
COMPUTATIONAL	LABEL	SUBTRACT
COMPUTATIONAL-1	MEMORY	SYNCHRONIZED
CONFIGURATION	MOVE	VALUE
DATA	MULTIPLY	WORKING-STORAGE
DISPLAY	OBJECT-COMPUTER	WRITE
DISPLAY-1	OCCURS	
DISPLAY-2	OPEN	
DISPLAY-3	PERFORM	
DISPLAY-4	PICTURE	
DIVIDE	PROCEDURE	

4 In messages 006, 011 and 012 the printed code nnnn refers to a list at the end of the error messages giving the COBOL keywords to which the messages refer (see page 234).

5 Certain messages are 'cautionary' only, i.e. they draw the user's attention to omissions in the source program which are nevertheless permitted or inconsequential. These messages have a (c) in the last column in the list, and print the word CAUTION instead of ERROR.

6 Error message 027 – GIVING ALONE

This message will most commonly occur when the user has written an invalid ADD statement of the form:

ADD ... TO ... GIVING ...

The compiler processes this as a valid ADD ... TO ... statement, but then finds a 'hanging' GIVING ... clause, which is flagged as being ALONE.

Other messages printed in the Analysis phase which include the word ALONE often result from similar errors in the COBOL source, – either the statement preceding the keyword printed in the message is self-sufficient and complete, or part of the context of the keyword has been found to contain other errors and has been ignored so leaving the quoted keyword, in effect, 'hanging'.

7 Error message 303 – LITERAL ASSOCIATED WITH FIELD OF DIFFERENT CLASS.

This message may be the result of an error in a PICTURE clause in the Data Division.

If, for example, the programmer writes

02 ITEM-N PICTURE X(300)

(in error because the size exceeds 120 characters), the compiler will print message 141 INVALID PICTURE (G) and will assume a DISPLAY NUMERIC field one character in size.

If then the programmer has written a Procedural statement such as

IF ITEM-N EQUAL TO SPACES ...

error message 303 will be printed because a numeric field cannot be compared with a non-numeric literal such as SPACES.

8 If any of the errors flagged by the messages below to page 204 occur, compilation will halt at the end of the current compiler overlay, displaying the message SERVICED (etc.) on the console typewriter.

Format and parameter errors

Printer message	Cause	Programmer action
A1 NO ENVIRONMENT DIVISION A2 NO DATA DIVISION A3 NO PROCEDURE DIVISION	Either the specified line has been omitted from the source program, or been mispunched, or an End Marker has appeared in the middle of the program.	Insert required line in source, or extract offending End Marker, and recompile.
A4 (unused)		
A5 TOO MANY PROCEDURE NAMES	As printed. See also page 270.	The number of procedure names must be reduced, or on multi-programming machines more store must be allocated to the compiler if possible before the program can be compiled.
A6 (unused)		
A7 (unused)		
A8 TOO MANY NAMES	There are too many data and/or procedure names in the source program. See also page 270.	Rewrite, reducing number of names in source program, or find the compiler in more store.
600 LONG BLOCK	A block in the source program on paper tape has more than 80 characters + NEWLINE, *or* more than 40 shift characters, *or* more than 3 tabulations, (*CORRECT run).	Repunch the offending block and recompile.
601 LONG BLOCK	As above, but only printed during a *COMPILE run.	As above.
602 LONG BLOCK	As above, but refers to a Correction line on paper tape, (*CORRECT run).	As above.
603 (unused)		
604 LONG BLOCK(S)	Correction line blocks on paper tape have been found with more than 80 characters + NEWLINE, or more than 40 shift characters, or more than 3 tabulations, or illegal characters. Always printed if there have been any 602 messages or 615 messages.	Repunch the offending Correction line blocks and recompile.
605 LONG BLOCK(S)	As above, referring to source program blocks on paper tape. Always printed if there have been any 600 messages or 616 messages.	Repunch the offending blocks and recompile.

Printer message	Cause	Programmer action
606 (unused)		
607 AMENDMENTS OUT OF SEQUENCE	As printed.	Correct and recompile.
608 (unused)		
609 (unused)		
610 LONG BLOCK	A Steering line on paper tape has more than 80 characters + NEWLINE in the block.	Correct the Steering line block and recompile.
611 STEERING LINE: INVALID DIRECTIVE	As printed.	Correct and recompile.
612 STEERING LINE: INVALID CHARACTERS(S)	As printed.	Correct and recompile.
613 STEERING LINES: NO MAJOR DIRECTIVES READ	As printed.	Insert or amend Steering lines and recompile.
614 LONG BLOCK	A Rapidwrite source program block on paper tape has more than 80 characters + NEW-LINE, *or* more than 40 shift characters.	Repunch the offending block and recompile.
615 ILLEGAL CHARACTER(S)	A Correction line on paper tape has an illegal character (see note on page 203).	Correct and recompile.
616 ILLEGAL CHARACTER(S)	A COBOL source line on paper tape has an illegal character (occurs in *CORRECT run). See note on page 203.	Correct and recompile.
617 (unused)		
618 (unused)		
619 TOTAL OF nnnn ERROR(S) SIGNALLED	As printed.	Correct errors and recompile (refers to messages already flagged).
620 O/P LOWER nnnnnn WORDS TOO LARGE	The Lower Data Area of the object program has been filled. See also page 269.	Reduce size of data areas and recompile.
622 SERIOUS ERROR(S) AS ABOVE	Errors as detailed in previous messages have been detected during Consolidate phase.	Correct errors and recompile.
623 (see page 238)		
624 (see page 238)		
625 (see page 238)		
626 (see page 238)		
627 (see page 238)		
628 TOO MANY FILES IN STEERING LINES	As printed (compiler limit at present 13). See also page 269.	Reduce number of files and recompile.
629 (see page 238)		
630 (see page 238)		

Printer message	Cause	Programmer action
631 NO STEERING LINE *CONSOLIDATE 632 NO STEERING LINE *COMPILE 633 NO STEERING LINE *COBOL 634 NO STEERING LINE *RAPIDWRITE	Inconsistent Steering lines, e.g. *CONSOLIDATE without *COMPILE	Supply consistent Steering lines and recompile.

Note: The permitted COBOL Character Set and Data Character Set are given in Appendix 1. If an illegal character is read in a source program or Correction line on paper tape, an error is flagged (see above) and the character is replaced for compilation either by #75, or by two characters: #76 followed by a character which will vary according to the mispunching.

Printer message	Cause	Programmer action
635 COMPILER LIMITS EXCEEDED	The size of the source program, reflected in compiler-generated tables, is too large for the core store available to compilation. See also page 269.	The number of data and/or procedure names must be reduced, or more core store made available, before recompiling.
636 INVALID 'MT' OPERAND FOR *AMENDMENTS	Correction lines, according to the Steering line, are to be read from magnetic tape.	Supply new Steering line assigning Correction lines to cards or paper tape.
637 STEERING LINE: INVALID SYNTAX	Various format errors in one of the Steering lines.	Supply corrected Steering line and recompile.
638 INVALID 'MT' OPERAND FOR *RAPIDWRITE	According to the Steering line a Rapidwrite source program is to be read from magnetic tape.	Supply new Steering line assigning the Rapidwrite source to cards or paper tape.
639 NO STEERING LINE *OBJECT	As printed.	If the Steering lines *COMPILE and *CONSOLIDATE are used, *OBJECT must also be present: Supply and recompile.
640 INVALID M/T SOURCE	A COBOL input source on magnetic tape is not in the correct format, or has parity errors.	Replace by tape in correct format, or recreate tape, and recompile.
641 ILLEGAL CHARACTER	An illegal character has been read in a block of Rapidwrite source on paper tape (see note below).	Repunch the offending block and recompile.
642 ILLEGAL CHARACTER	An illegal character has been read in COBOL source input (see note below). Occurs in a *COMPILE run.	As above.
643 (see page 238)		
644 COMPILER LIMITS EXCEEDED	A cue table of "forward references" made up and used by the compiler is full. See also page 270.	Reduce the number of "forward references" in the source program and recompile.
645 COMPUTER SIZE EXCEEDED	The size quoted in the OBJECT-COMPUTER clause has been exceeded. See also page 270.	If possible more core store should be assigned in the OBJECT-COMPUTER clause. Otherwise the source program may have to be segmented or split into two or more programs.

Printer message	Cause	Programmer action
646 (unused)		
647 (unused)		
648 (unused)		
649 (unused)		
650 COMPILER NEEDS MORE MAGNETIC TAPES	Too many magnetic tape files required for the compilation. See also page 270.	Reduce the number of magnetic tape files, or specify more magnetic tapes as available to the System and recompile.
651 to 660 (see pages 238 and 239)		
661 NO STEERING LINE *BINARY FOR O/LOAD OBJECT PROGRAM	As printed.	The object program may be consolidated by supplying the appropriate slow peripheral parameters to XPCL.
662 (see page 239)		
664 AUXILIARY CONTROL AREA FULL	As stated.	Rectify the fault and rerun.
665 UNCLEARABLE PARITY ERROR	As stated.	Reallocate the file named in 662 console message, and rerun.
666 LOGICAL BUCKET NUMBER OUT OF RANGE OR ZERO	The file named in the 662 message on the console typewriter is too small.	Rectify the fault and rerun.
668 NO SPACE AVAILABLE TO EXTEND A FILE OR OPEN A SCRATCH FILE	As printed.	Recompile using another cartridge or supply appropriate permanent files.
670 FILE NOT IN SYSTEM	As printed.	Recompile supplying missing file (mispunched steering line?)
671 FAILURE OF INTEGRITY CODE CHECK	(a) File was already in use by another program when the compiler tried to open it, *or* (b) incorrect use has been made of the highest generation facility.	Recompile.
672 ON OPENING FILE NO SPACE AVAILABLE FOR FILE DESCRIPTION	May occur in a dual or multiprogramming environment.	Recompile after a program using E.D.S. is deleted.
673 PURGE DATE NOT EXCEEDED	As listed.	Recompile using a different file or after using XJEX to zeroize the data retention period on the file.
674 UTILITY ROUTINE WRITING TO SYSTEM CONTROL AREA AS COMPILER ATTEMPTED TO OPEN FILE	As listed.	Recompile when no E.D.S. utility is in action.

Printer message	Cause	Programmer action
675 FILE filename (generation number) HAS BEEN {EXTENDED / CONTRACTED} BY nnnn BLOCKS 676 FILE PROGRAM NAME (name) HAS BEEN {EXTENDED / CONTRACTED} BY nnnn BLOCKS	As stated. nnnn will always be a multiple of 80, and in the case of a CONTRACT message will be the same as in the preceding 675 EXTEND message relating to this file. If the size of any of the extensions was increased by Executive, then on an EXTEND message the figure nnnn will be inaccurate, and the corresponding contraction will not necessarily be complete.	None
677 WK FILE filename (FGN nnn) UNFOUND	An informatory message not involving recompilation.	None
678 SOURCE TAPE INVALID FORMAT 679 SOURCE SUBFILE NOT FOUND	Refers to either the source file or the subroutines file.	Check that the correct tape or disc file was opened. If so, send a dump of the appropriate file and other relevant information, to International Computers Limited

Note: When illegal characters are flagged by messages 641 and 642, they are ignored and no characters are passed on in their place.

COPY phase errors

Printer message	Cause	Compiler action
C01 PERIOD MISSING	Full stop is missing between paragraph or section, and the word COPY.	The presence of a full stop is assumed (the message is a CAUTION only).
C02 INVALID LIBRARY-NAME	As printed.	The COPY statement is ignored.
C03 INVALID COPY PARAGRAPH	Other statements apart from COPY in the sentence or paragraph.	The other statements are ignored.
C04 INVALID COPY CLAUSE	Full stop or REPLACING missing after library entry-name in COPY statement.	The rest of the statement is ignored.
C05 INVALID REPLACING PARAMETER	As printed.	The rest of the statement is ignored.
C06 BY MISSING	BY MISSING after REPLACING parameter.	The rest of the statement is ignored.
C07 INVALID FILE/RECORD NAME	As printed.	Assumes that file or record declaration is not present.
C08 COPY OUT OF CONTEXT	As printed.	The COPY statement is ignored.
C09 INVALID SEGMENT-LIMIT	Invalid parameter following SEGMENT-LIMIT IS.	The clause is ignored.
C10 INVALID SECTION DECL	Full stop or priority number missing after section declaration.	The section declaration is ignored.
C11 TOO MANY LIBRARY-NAMES	The compiler limit on the number of library entry-names that may be referenced has been exceeded.	The COPY statement is ignored.
C12 ENTRY NOT IN LIBRARY	As printed.	The COPY statement is ignored.

Printer message	Cause	Compiler action
C13 TOO MANY SECTION-NAMES	As printed (compiler limit exceeded).	The section is treated as part of preceding section.
C14 PRIORITY-NO>99	As printed.	A priority-number of 99 is assumed.
C15 SEGMENT-LIMIT>49	As printed.	Segment-limit is assumed to be 49.
C16 LITERAL>58 CHARACTERS	As printed.	Only the first 58 characters are accepted.

COBOL language errors

Message	Cause	Compiler action
001 PROG-ID INVALID/MISSING	*Either* (a) the PROGRAM-ID statement is missing or different to the name quoted in the *IDENTITY steering line, *or* (b) the name quoted in the *IDENTITY steering line for a master program contains more than 4 characters, *or* (c) the name quoted in *IDENTITY steering line for a subroutine contains more than 11 characters.	(a) The name quoted in the *IDENTITY steering line is taken. If there is no such steering line the name COBL 01 is assumed. (b) The first 4 characters only are taken. (c) The first 11 characters are taken.
002 DUPL. LEVEL NO	There is more than one valid level number at the beginning of a data declaration sentence.	The first level number is taken as correct; the others are ignored.
003 UNDEFINED NAME D....	A data name referenced in the Procedure Division has not been defined in the Data Division.	D.... is the name itself (up to 30 characters). The data name is ignored; this may lead to the whole sentence being ignored.
004 INVALID LEVEL NO	A field level number is greater than 49 and not equal to 66, 77 or 88.	The level number is ignored; this may lead to error message 008 being printed as well.
005 DUPL. NAME D.... nn	The same data name has been used to define more than one field in the Data Division.	D.... is the name (up to 30 characters). nn is the number of duplications. The first occurrence of the name is assumed to be the correct one.
006 EMPTY LIST nnnn	The parameters required by a keyword are missing or incorrect.	The keyword will be ignored. For nnnn see Code List on page 234.
007 PERIOD MISSING OR EXTRA PARAMETER	A full stop is missing before a heading in the Env or Data Division *or* there is a superfluous operand at the beginning of a sentence.	The heading or the valid part of the sentence is processed: other parts of the context may be ignored - will lead to unpredictable results.

Message	Cause	Compiler action
008 LEVEL-NO or DATA-NAME INVALID/MISSING	*Either* a valid level number *or* a valid data name is missing at the beginning of a data declaration line.	The whole data declaration line is ignored.
009 FD/FILLER OUT OF CONTEXT	A valid data name is missing at the beginning of a data declaration sentence.	The sentence is ignored.
010 PRIORITY INVALID/ MISSING – 01 ASSUMED	As printed.	The priority of the source program is assumed to be 01.
011 TOO FEW OPERANDS nnnn	Some of the operands of a keyword are missing.	The statement containing the error is ignored. For nnnn see Code List on page 234.
012 INVALID SYNTAX nnnn	The keyword being processed has incorrect operands.	As above.
013 SENTENCE TOO LONG/ COMPLEX	There are too many statements in the sentence, or the sentence is too complex (compiler limits as yet undefined). See also page 265.	The statement being processed is ignored.
014) ALONE	A close parenthesis is encountered without a preceding open parenthesis.	The close parenthesis is ignored and processing of the statement is resumed.
015 NO RERUN	The keyword TAPE-* or REEL is encountered without RERUN preceding it.	The statement being processed is ignored.
016 LONE ACTIVE-TIME	ACTIVE-TIME is out of context.	The statement being processed is ignored.
017 AFTER ALONE	AFTER is out of context.	The statement being processed is ignored.
018 AT ALONE	AT is out of context.	The statement being processed is ignored.
019 BY ALONE	BY is out of context.	The statement being processed is ignored.
020 NO BLOCK	CHARACTERS or RECORDS is encountered without BLOCK preceding it.	The presence of BLOCK is assumed. (c)
021 DEPENDING ALONE	DEPENDING is out of context.	The statement being processed is ignored.
022 DIVISION ALONE	DIVISION is out of context.	The statement being processed is ignored.
023 LONE END	END is out of context.	The statement being processed is ignored.
024 ERROR ALONE	ERROR is out of context.	The statement being processed is ignored.
025 FROM ALONE	FROM is out of context.	The statement being processed is ignored.
026 GENERATION NO ALONE	GENERATION NO is out of context.	The statement being processed is ignored.
027 GIVING ALONE	GIVING is out of context.	The statement being processed is ignored.

Message	Cause	Compiler action
028 ID ALONE	ID or IDENTIFICATION is out of context.	The statement being processed is ignored.
029 NO OPEN	INPUT or OUTPUT is encountered without OPEN preceding it.	The statement is processed assuming the presence of OPEN. (c)
030 INTO ALONE	INTO is out of context.	The statement being processed is ignored.
031 NO SYNCHRONIZED	RIGHT or LEFT encountered without SYNCHRONIZED.	The statement is processed assuming the presence of SYNCHRONIZED.
032 MODE ALONE	MODE is encountered without RECORDING preceding it.	The statement is processed assuming the presence of RECORDING. (c)
033 NO ASSIGN	OBJECT-PROGRAM is encountered without ASSIGN preceding it.	The statement is processed assuming the presence of ASSIGN. (c)
034 OF ALONE	OF is out of context.	The statement being processed is ignored.
035 NO LABEL	OMITTED or STANDARD is encountered without LABEL RECORDS preceding it, *or* RECORDS is encountered without LABEL preceding it.	The statement being processed is ignored. The statement is processed assuming the presence of LABEL. (c)
036 NO ALTER	PROCEED is encountered without ALTER preceding it.	The statement being processed is ignored.
037 NO SAME	RECORD is encountered without preceding SAME.	The statement being processed is ignored.
038 NO INCLUDE	REPLACING is encountered without INCLUDE preceding it.	The statement being processed is ignored.
039 LONE RWD	REWIND is encountered without NO preceding it.	The statement is processed assuming the presence of NO.(c)
040 LONE ROUNDED	ROUNDED is out of context.	The statement being processed is ignored.
041 NO STOP	RUN is encountered without STOP preceding it.	The statement is processed assuming the presence of STOP.(c)
042 LONE SECTION	SECTION is out of context.	The statement being processed is ignored.
043 SIZE ALONE	SIZE is out of context.	The statement being processed is ignored.
044 TAPES ALONE	TAPES is out of context.	The statement being processed is ignored.
045 THRU ALONE	THRU is out of context.	The statement being processed is ignored.
046 TIMES ALONE	TIMES is out of context.	The statement being processed is ignored.
047 TO ALONE	TO is out of context.	The statement being processed is ignored.
048 BITS ALONE	BITS is out of context.	The statement being processed is ignored.

Message	Cause	Compiler action
049 NO MEMORY	WORDS is encountered without MEMORY preceding it.	The statement is processed assuming the presence of MEMORY. (c)
050 ZERO ALONE	ZERO is out of context.	The statement being processed is ignored.
051 ASSIGN ALONE	ASSIGN is out of context.	The statement being processed is ignored.
052 NO TO	TO is missing after one of the keywords EQUAL, GO or MOVE.	The statement is processed assuming the presence of TO.(c)
053 NO)	) is missing after (.	The statement is processed assuming the presence of). (c)
054 NO ERROR	ERROR is missing after SIZE.	The statement is processed assuming the presence of ERROR. (c)
055 NO SIZE	ERROR is encountered without SIZE preceding it.	The statement is processed assuming the presence of SIZE.(c)
056 NO UNTIL	UNTIL missing in a PERFORM ... VARYING ... statement.	The presence of UNTIL is assumed. (c)
057 NO ZERO	ZERO is missing after BLANK.	The statement is processed assuming the presence of ZERO.(c)
058 BLOCK ALONE	BLOCK is encountered without CHARACTER or RECORDS following it.	The statement is processed assuming the presence of CHARACTERS. (c)
059 DIVIDE ALONE	DIVIDE is encountered with INTO missing.	The statement is processed assuming the presence of the missing word. (c)
060 NO SECTION	SECTION is missing after one of the keywords FILE or WORKING-STORAGE.	The statement is processed assuming the presence of SECTION. (c)
061 LABEL ALONE	LABEL is encountered without OMITTED, STANDARD or RECORDS following it.	The statement is processed assuming the presence of STANDARD. (c)
062 NO WORDS	MEMORY is encountered without WORDS following it.	The statement is processed assuming the presence of WORDS. (c)
063 MULTIPLY ALONE	MULTIPLY is encountered with BY missing.	The statement is processed assuming the presence of the missing word. (c)
064 NO REWIND	REWIND is missing after NO	The statement being processed is ignored.
065 (unused)		
066 DATA ALONE	DATA is encountered with neither DIVISION nor RECORDS following it.	The statement being processed is ignored.
067 READ ALONE	READ is encountered with either AT or END missing.	The statement is processed assuming the presence of the missing word.(c)
068 (unused)		

Message	Cause	Compiler action
069 NO REEL	REEL is missing after RERUN.	The statement is processed assuming the presence of REEL. (c)
070 NO RECORD	RECORD is missing after SAME.	The statement is processed assuming the presence of RECORD. (c)
071 NO ASSIGN	ASSIGN is missing after SELECT.	The statement is processed assuming the presence of ASSIGN. (c)
072 NO FROM	FROM is missing after SUBTRACT or there is a format error in a PERFORM... VARYING... AFTER statement, e.g. VARYING is repeated.	In the first case FROM is assumed; in the second case invalid coding may be produced for the PERFORM statement.
073 SYNCH. ALONE	SYNCHRONIZED is encountered with neither LEFT nor RIGHT following it.	SYNCHRONIZED is ignored and next word is processed.
074 LONE NOT	NOT is encountered without EQUAL, GREATER or LESS following it.	The statement being processed is ignored.
075 NO DIVISION	DIVISION is missing after PROCEDURE.	The statement is processed assuming the presence of DIVISION. (c)
076 NO ID	ID or IDENTIFICATION is missing after VALUE OF.	The statement is processed assuming the presence of ID. (c)
077 LONE RESERVE	RESERVE is out of context.	The statement being processed is ignored.
078 INVALID WORD IN COL.8	As printed: occurs in Environment and Data Divisions; also may be printed if a Data Division field-definition is not terminated by a full stop.	In the first case the word is ignored; in the second the whole data definition may be ignored.
079 SENTENCE INVALID FOR THIS DIVISION	Sometimes caused by the presence of superfluous operands in a sentence, sometimes by the omission of a DIVISION header.	The sentence is ignored.
080 SEQUENCE ERROR	As printed. (Rapidwrite input.)	The offending sequence number is nevertheless taken as the basis of comparison with the number of the next line. The source line is processed.
081 PROGRAM IDENTIFICATION INCORRECT	As printed. (Rapidwrite)	As above, re identification.
082 INVALID ENVIRONMENT CARD	Columns 13 and 14 of an Environment Division card do not contain SO, OB, AS or DU. (Rapidwrite)	The card is ignored.
083 INVALID HARDWARE NAME	Column 62 of an Environment Division OB card does not contain a space, C or P; *or* columns 36 and 37 of an AS card do not contain +C, +P, +M or +W. (Rapidwrite)	In the first case, an ASSIGN OBJECT-PROGRAM clause is not generated; in the second, no hardward name is generated.

Message	Cause	Compiler action
084 VALUE OF ID TOO LONG	The value of the identification on an FD card is more than 12 characters. (Rapidwrite)	The first 12 characters are accepted as the identification.
085 LABEL RECORDS CLAUSE INVALID	Column 38 of an FD card does not contain a space, S or O. (Rapidwrite)	LABEL RECORDS STANDARD is generated.
086 VALUE TOO LONG	The value is more than 120 characters. (Rapidwrite)	Only the first 120 characters are accepted as the value.
087 INVALID SYNCHRONIZED CLAUSE	Column 44 of a Data Division card does not contain a space, L or R. (Rapidwrite)	SYNCHRONIZED RIGHT is generated.
088 PICTURE TOO LONG	The picture contains more than 30 characters. (Rapidwrite)	Only the first 30 characters are accepted as the picture.
089 INVALID VERB CODE	Columns 6 and 7 of a Procedure Division card do not contain a permitted verb code. (Rapidwrite)	The card is ignored.
090 INVALID FORMAT	Box 4 of an IF card does not contain =, <, >, NOT=, NOT<, or NOT>, *or* box 3 of an OP card does not contain I or O, *or* box 5 of a WR card does not contain spaces, A or B, *or* box 6 of a CO card does not contain +, -, * or /, *or* box 4 of an MO card contains + but not +K1, *or* box 6 of an MO card contains + but not +K1 or +ALLn. (Rapidwrite)	In all cases, the card is ignored.
091 INVALID QUALIFICATION	(Rapidwrite input.) The qualifier is not a valid file name.	The qualifier is ignored.
092 ILLEGAL NAME OR HEADING	There is an illegal word beginning in column 8, i.e. a word which is not 'FD', or a level number, or a Header, or a section or paragraph declaration. Alternatively, an illegal procedure-name has been quoted in a GO or PERFORM statement.	The word is passed on but will lead to other error messages referencing the sentence in which it occurs. If it is an illegal procedure name, later references to it, e.g. in GO or PERFORM statements, will produce invalid coding.
093 WORD GREATER THAN 30 CHARS	As printed, refers to a data or procedure name or to the number of characters in a Picture.	Only the first 30 characters are accepted as the name
094 INCORRECT IDENTIFICATION	The identification in columns 73 to 80 of the current lines differs from that in the first line of the source program.	No effect on compilation.
095 (unused)		
096 (unused)		
097 SEQUENCE ERROR	As printed.	The line is processed as if correct.
098 WORD GREATER THAN 120 CHARS	As printed, refers to a non-numeric literal.	Only the first 120 characters of the literal are accepted.

Message	Cause	Compiler action
099 MISSING QUOTES CHAR	The open quote or close quote of a non-numeric literal is missing.	The next 120 characters in the source program including spaces are taken as a literal, probably mutilating the following statement.
100 COMPILER STORAGE EXCEEDED: PICTURE/ VALUE IGNORED	The complexity of the field description is such that a compiler limit, as yet undefined, has been exceeded. See also page 266.	The picture or value of the field will be ignored and object coding referencing the field will be invalid. *Remedy:* split the field into subfields.
101 COMPILER STORAGE EXCEEDED: INCORRECT OBJECT CODE	There are too many files in the source program (maximum permitted – 64). See also page 265.	All references to current and following files in the Procedure Division will generate incorrect object coding.
102 FILE INF CLASH	(a) Magnetic tape files: either VALUE OF ID clause is missing (the file identity) or the file identity is longer than 12 characters. (b) Slow peripheral files: various clauses (e.g. VALUE OF ID, LABEL RECORDS) have been used which apply only to magnetic tape or other fast peripheral files.	(a) In the first case, a file identity WORKTAPE and retention period zero are assumed; in the second, the first 12 characters are taken as the identity. (b) The clauses are ignored.
103 FIELD TOO LARGE	A field OCCURS more than 262,143 times *or* a table as defined would require more than 349,525 words of storage. See also page 266.	All references to the table elements will generate invalid coding.
104 BLOCK SIZE WRONG	(a) Any peripheral: the size of the largest record in the file exceeds maximum permitted (e.g. 80 characters for card input file). (b) Card punch or printer: size of record is not equal to 80 or 120 characters respectively.	In all cases, there is no corrective action, i.e. the user-defined size is accepted. (c)
105 (unused)		
106 FILE UNASSIGNED	SELECT ... ASSIGN statement is missing in Environment Division for a file defined in the Data Division.	The file is assigned to Magnetic Tape Unit 0 and the object program probably cannot be run successfully.
107 INVALID OCCURS	*Either* a record (01 level) has been described as OCCURring (as a table item), *or* a table has more than 3 levels.	In both cases, the current OCCURS statement is ignored.

Message	Cause	Compiler action
108 WRONG REDEFINES	*Either* a 01 level in the File Section is being redefined, *or* a redefining field or a subfield of the redefining field has an initial value, *or* the redefined and redefining fields have different levels, *or* they are not consecutive at their level, *or* they have different sizes (at levels other than 01).	In the first four cases, the redefining clause is ignored. In fifth case, store is allocated equivalent to the larger of the two fields. The message will also be printed if a redefining record (01 level) is larger than that redefined and all Lower allocation has been used. The redefining record will then go into Upper and any part of the user's program that expects the two records to share the same area will be invalid.
109 NAME NOT PREVIOUSLY DEFINED	data-name in an OCCURS... DEPENDING clause has not yet been declared in the Data Division	The OCCURS...DEPENDING clause is treated as an occurs clause.
110 LEVEL ERROR	(a) Level > 49 and ≠88 or 77 or 66. (b) Non-elementary field with no subfields. (c) First field after File Description or Working Storage Section not 01 level. (d) A variety of level errors, i.e. levels which are found to be incorrect only by comparison with preceding or following levels.	(a) Level of 49 is assumed. (b) Next field is made a subfield unless its level number would then be > 49 when it is ignored and its description attached to this field. (c) 01 is assumed. (d) The compiler makes the minimum adjustment necessary to compose a valid sequence of levels.
111 VARIABLE FIELD MISPLACED	An OCCURS...DEPENDING clause appears in a field which is neither the last elementary field in the record nor contains the last elementary field.	No compiler action.
112 USAGE CLASH	The USAGE specified is incompatible with the Picture.	A Usage compatible with the Picture is assumed.
113 ILLEGAL VALUE	*Either* a value has been assigned to a file in the File Section, *or* in an occurring field or in a subfield of an occurring field, *or* characters in the value are incompatible with the field's class and usage, *or* a value has been assigned to a field in the LINKAGE SECTION.	In all cases the value is ignored.
114 NOT ENOUGH LOWER	See page 243.	See page 241.
115 F RECORDS NOT SAME LENGTH	Records are not of the same length in a file described as RECORDING MODE F	The object program may contain invalid coding
116 VARIABLE FIELD IN F RECORD	A variable field appears in a record of a file described as RECORDING MODE F	The OCCURS...DEPENDING clause is treated as an OCCURS clause
117 OBJECT COMPUTER MEMORY INVALID/ MISSING	*Either* no OBJECT COMPUTER MEMORY... WORDS clause in Environment Division, *or* the memory size specified is out of range.	The memory size is taken as 8K.

Message		Cause	Compiler action
118	OBJECT COMPUTER NAME INVALID/ MISSING	*Either* no OBJECT COMPUTER name is given, *or* the name is invalid.	1902 is assumed.
119	SEGMENT-LIMIT INVALID	The segment limit is not a legal integer, *or* = 0, *or* is > 49.	A segment limit of 49 is assumed.
120	> 1 VARIABLE FIELD IN RECORD	As printed	OCCURS ...DEPENDING fields after the first are treated as OCCURS field.
121	FILE CAUSING DUMP IS UNDEFINED	Invalid RERUN statement.	The Controlling file is assigned to Magnetic Tape Unit 1.
122	NUMBER OF RECORDS CAUSING DUMP INVALID	As printed.	The number is assumed to be 16,000.
123	MORE THAN ONE RERUN CLAUSE FIRST ONLY CONSIDERED	As printed.	All but the first clause are ignored.
124	VARIABLE FIELD WITHIN TABLE	A variable field is also described by an OCCURS, or is within a group field described by an OCCURS	The OCCURS...DEPENDING clause is treated as an OCCURS clause
125	(unused)		
126	PHRASE CONFIGURATION SECTION INVALID/ MISSING	As printed.	No effect on compilation.
127 to 129 (unused)			
130	PHRASE INPUT-OUTPUT SECTION INVALID/ MISSING	As printed.	No effect on compilation.
131	PHRASE FILE-CONTROL INVALID/MISSING	As printed.	No effect on compilation.
132	(unused)		
133	(unused)		
134	PHRASE FILE SECTION INVALID/MISSING	As printed.	No effect on compilation.
135	SYNCHRONIZED CLASH	Both LEFT and RIGHT have been specified for the same field.	(a) The field is right synchronized. (b) Extra characters of store may be allocated which might, for example, make a redefining field too large. (c) The results of operations referring to the field may then be incorrect.
136	NEEDS FILLER (A)	The record as it stands would lead to incorrect data allocation (e.g. character field not starting on character boundary).	The record is adjusted by insertion of fillers. (c)
137	NEEDS FILLER (S)	As 136, referring to occurring field.	As above. (c)
138	COMPILER STORAGE EXCEEDED: TOO MANY FILES	There are more than 64 files in the source program. See also page 266.	Current and subsequent files are ignored; all following records will be assigned to the last file accepted.

Message	Cause	Compiler action
139 COMPILER STORAGE EXCEEDED: RECORD SIZE ALTERED	As printed or there are too many and/or complex fields in a record (compiler limit undefined). See also page 266.	The current field is taken as a record (01 level) and subsequent fields and records are processed as written, thus invalidating object coding referencing this record
140 (unused)		
141 INVALID PICTURE (letter)	The letter in the message will be one of the following, with the given interpretation: A – illegal character. B – a BLANK WHEN ZERO clause conflicts with suppression/ insertion symbols in the PICTURE of the field. C – misplaced report sign or illegal combination of characters, e.g. DR instead of CR. D – incompatible characters (e.g. £1 or if V appears in the same PICTURE as a period when *STANDARD is in force). E – the physical size of the Picture is > 30 characters. F – the point position is out of range. G – data size = 0, or is greater than the permitted maximum for a particular usage (e.g. COMPUTATIONAL Field > 12 characters). H – FLOAT/SUPPRESS error (e.g. + + + Z). I – > 15 insertion symbols. J – bracket error.	A – The illegal character is ignored. B – The BLANK WHEN ZERO clause is ignored. C – The illegal character is ignored. D – The second character is ignored. E – No action (does not affect compilation). F – No action (does not affect compilation). G – The field is assumed to be non-elementary, for the purpose of MOVEs etc. H – The invalid float characters (e.g. Z in the example) are assumed to be '9's. I – No action until absolute number of editing parameters exceeds 32, when the excess parameters are ignored. J – Bracket is ignored.
142 SYNTAX ERROR HAS CAUSED WRONG MAPS	As printed: often the result of omitting a full stop at the end of a field description.	The store allocation for this record as shown in the Data Map is incorrect and object code addressing fields in the record may be invalid.
143 RECORD < 4 CHARS	The largest record in a file assigned to card or paper tape reader is < 4 characters in length: not an error but see page 133.	No corrective action.
144 to 165	(unused)	
166 INVALID SORT SYNTAX	The form of the SORT statement is invalid, i.e. keyword(s) out of context or missing or invalid operand(s)	The SORT statement is ignored
167 (unused)		
168 ILLEGAL LINKAGE SECTION	LINKAGE SECTION specified in a master program.	Treated as WORKING-STORAGE SECTION.
169 ILLEGAL FD	An FD level is specified in a subroutine.	No compiler action.
170 ILLEGAL PARAMETERS	The statement PROCEDURE DIVISION USING... is specified in a master program.	The USING... clause is ignored.

Message		Cause	Compiler action
171	NO LINKAGE SECTION	As printed; refers to a subroutine.	No compiler action.
172	TOO MANY SUBROUTINES	Too many CALL and ENTER statements in the source program (compiler limit undefined). See subroutine COBJ on page 252.	The statement is abandoned and code generated to call the run-time subroutine COBJ.
173	S/R NAME > 11 CHARS	The subroutine name in a CALL or ENTER statement contains more than 11 characters.	Only the first 11 characters are taken.
174	SUBSCRIPTED FILE	The subscripted name is a file.	References to this subscripted name will produce invalid coding, or will be ignored.
175	COMPILER LIMIT EXCEEDED	The current sentence is too long or complex. See also page 267.	The sentence is ignored.
176	ILLEGAL SUBROUTINE PARAMETER	A parameter in a CALL statement is anything other than an 01 or a 77 level.	A standard address is substituted for the parameter.
177	LONE PROGRAM	PROGRAM met without proceeding EXIT.	The statement being processed is ignored.
178	'GO' NOT LAST PHRASE	A GO TO which is not the last statement appears in the true branch of a condition or in the AT END branch of a READ, or the ON SIZE ERROR branch of an arithmetic statement.	Does not affect compilation.
179	TOO MAY SECTIONS	There are too many overlay segments in the source program; a variable compiler limit has been exceeded. See also page 267.	This and all following overlay segments will be included in the last segment accepted.
180	ILLEGAL SORT KEY LEVEL NUMBER	A key is a file or record name	The illegal key is ignored
181	INVALID ALTER	There is a missing operand in the ALTER statement.	The lone' operand – which needed the missing operand to make a valid pair – is ignored. If there is only one operand in the original ALTER statement, the whole statement is ignored.
182	DUPL. PROCEDURE NAME	The same procedure name has been used in more than one paragraph or section declaration.	All references to the procedure name in the source program will generate code referring to its first declaration.
183	ADVANCE LITERAL > 2	The 'advance' literal in COBOL WRITE ... AFTER ... statement is greater than 2. (Compact COBOL only)	The literal is changed to 2.
184	ADVANCE LITERAL	The 'advance' literal in WRITE... AFTER... statement contains a minus sign or a decimal point.	The literal is changed to 2.
185	ADVANCE FIELD UNDEFINED	The 'advance' field in WRITE... AFTER... statement has not been defined in the Data Division of the source program.	The ADVANCE field is changed to a literal of 2.

Message	Cause	Compiler action
186 (unused)		
187 >3 SUBSCRIPTS	More than 3 subscripts in the data name reference	Only the first 3 subscripts are accepted.
188 MISSING PARAGRAPH	A section declaration has not been followed by a paragraph declaration, or there is no paragraph declaration at the beginning of the Procedure Division.	No action.
189 NON SORT-FILE SORTED	The first filename quoted in a SORT statement is not an SD file	The SORT statement is ignored
190 UNDECLARED PARAGRAPH (name)	The quoted paragraph or section has never been declared in the source program. (No sequence number is printed with this message.)	The paragraph is declared at the end of the object program. Any reference to the paragraph will result at run-time in 'HALTED CZ'.
191 ILLEGALLY ALTERED PAR (name)	The quoted paragraph, when declared in the source program, was not immediately followed by a GO statement.	No corrective action: the ALTER statement will generate invalid coding.
192 ILLEGAL SUBSCRIPT nnnn	The subscript is a file name, *or* it is not a valid integer, *or* it is zero. (nnnn = subscripted name)	In the *first* case, the statement is ignored; in the *second* case, a numeric subscript as close to the original as possible is substituted; in the *third* case, the subscript is taken as 1.
193 NON-FILE READ (name)	The name is not a file name defined in Data Division.	The COBOL statement is ignored.
194 ILLEGAL PRIORITY	The section priority number is either (a) not a valid integer, or (b) less than the preceding one.	(a) A valid integer as close to the original as possible is substituted and used as the priority of all statements in this section. (b) A priority number equal to the preceding one is taken.
195 TOO MANY PROCEDURE NAMES	As printed: a compiler limit, as yet undefined, has been exceeded. See also page 267.	See page 201 message number A5.
196 LOCK/NO-REWIND IGNORED	As printed: this option not yet implemented.	The CLOSE statement is processed without the LOCK or NO REWIND option.
197 NON-RECORD WRITTEN (name)	The name is not a record name defined in the Data Division.	The WRITE statement is ignored but if it includes the AFTER option, the advance field is nevertheless checked for error.
198 NON-FILE OPENED (name)	The name is not a file name defined in Data Division.	The COBOL OPEN statement is ignored, so far as the reference to this name is concerned.
199 NON-FILE CLOSED (name)	The name is not a file name defined in Data Division.	The COBOL CLOSE statement is ignored, so far as the reference to this name is concerned.

Message	Cause	Compiler action
200 LOWER STORAGE EXCEEDED BY nnnn WORDS 200 LOWER STORAGE EXCEEDED	See page 243.	Generation of code ceases.
201 OUTPUT FILE nn ON CR (or TR) 201 INPUT FILE nn ON LP (or CP, TP)	Attempt to READ a file with an output device assigned to it, and other similar errors.	The statement is ignored.
202 (unused)		
203 ADVANCING DATA NAME INVALID	Data name in ADVANCING clause is non-integral.	A literal of 2 is assumed.
204 ADVANCING NOT ON PRINTER	WRITE statement for a record of a file not assigned to printer has an ADVANCING clause.	The ADVANCING clause is ignored.
205 WRITING WORKING STORAGE RECORD	As printed; (also applies to records in the CONSTANT and LINKAGE section).	No code is generated for the statement.
206 (unused)		
207 ENTER: TOO MANY INCLUDED ROUTINES	As printed. See subroutine COBJ page 252 and also page 267.	The statement is abandoned and code generated to call the run-time subroutine COBJ.
208 ENTER: S/R NAME MORE THAN 11 CHARACTERS	A maximum of 11 characters can be handled.	The first 11 characters only are taken.
209 DISPLAYS ONLY 40 CHARACTERS	The maximum length of a message of the console is 40 characters.	The first 40 characters only are displayed.
210 COMPUTER SIZE EXCEEDED BY nn WORDS	The total number of words required for data storage, permanent program and maximum overlay is greater than the object computer size. See also page 268.	None.
211 STOP MESSAGE MORE THAN 40 CHARACTERS	The maximum length of a message at the console is 40 characters.	The first 40 characters only are taken as the message.
212 CLASS: NON-CHARACTER FIELD	Class test refers to an elementary field defined as COMPUTATIONAL or COMPUTATIONAL-1, or to a group field which is not 'quasi-character' (see page 199).	The statement is ignored, i.e. object code is generated so that the test always 'fails'.
213 CLASS: INCOMPATIBLE TEST	IF....NUMERIC test refers to an alphabetic field or IF.... ALPHABETIC refers to a numeric field.	The statement is ignored, i.e. object code is generated so that the test always 'fails'.
214 (unused)		
215 NON SORT FILE RETURNED (name)	as stated	The RERUN statement is ignored

Message	Cause	Compiler action
216 NON-RECORD RELEASED (name)	as stated	The RELEASE statement is ignored
217 and 218 (unused)		
219 NAME TREATED AS CONDITION-NAME	(See page 71)	As stated
220 to 301 (unused)		
302 ENTER: PARAMETER nn HAS BIT ADDRESS	One of the parameters in an ENTER statement is a non-synchronized COMPUTATIONAL or COMPUTATIONAL-1 field, or a group field which is not an integral number of characters long and does not have a character.address.	The bit address is provided at object time (e.g. if a field started on the fourth bit of the third character of the seventh word of Lower Preset, its bit address would be 159 relative to the start of LP). (c)

Message	Cause	Compiler action
303 LITERAL ASSOCIATED WITH FIELD OF DIFFERENT CLASS	The literal contains a character which is invalid with respect to the field in which it is to be stored (see page 15 for rules, and note that if the field is sterling, the literal character corresponding to the 10/- digit must not exceed 1).	If the field is COMPUTATIONAL, the offending character(s) are ignored and the gap closed up with zero fill on the left. If the field is COMPUTATIONAL-1, the least significant bit of the character is taken when the literal is alphanumeric. Zeros are assumed when the literal is SPACES; if the field is non-elementary and not quasi-character, the only permitted literal is ZEROS (which will be assumed).
304 NOT ALL LITERAL CHARACTERS MAP ON TO FIELD	After aligning the literal it is found that certain characters have to be truncated. e.g.If the literal is 3.25 and the target field PICTURE is 9V9, the '5' is truncated. If the literal is 12 and the target field PICTURE is 99PPP, there is no overlap at all and both digits are truncated (to move 12 to such a field the literal must be 12000).	Truncation takes place.
305 FIELDS DO NOT OVERLAP	There are no common digit positions between the two fields concerned, e.g. MOVE A TO B when A's PICTURE is 999 and B's PICTURE is V99 *or* ADD A B GIVING C when the PICTUREs are: A – 99P B – 99P C – 9 (illustrating the case where none of the digits in the result of an arithmetic operation align with digits in the result field).	A value of zero is assumed and is stored in the target, with editing if required. If an arithmetic verb is involved, and it has an ON SIZE ERROR clause, the object coding will test if the result exceeds zero.
306 ROUNDING CLAUSE POINTLESS	The rounding clause is unnecessary since no truncation can occur.	The rounding clause is ignored.
307 SIZE-ERROR CLAUSE POINTLESS	The SIZE ERROR clause is unnecessary since the result field is larger than the maximum possible result.	Code may be generated for the statements in the SIZE ERROR clause, but they will never be obeyed at run time since SIZE ERROR cannot occur.
308 NEGATIVE LITERAL ASSOCIATED WITH UNSIGNED FIELD	The literal is preceded by a minus sign but the field is unsigned.	The modulus is taken.
309 (unused)		

Message	Cause	Compiler action
310 NON-CHARACTER GROUP FIELD	The data name is a file or a non-quasi-character group field or record.	The field is ignored.
311 ACCEPTED FIELD NON-ELEMENTARY	The field in an ACCEPT statement must be elementary.	The statement is ignored.
312 INVALID CONDITIONAL STATEMENT	*Either* an alphabetic, alphanumeric or edited field is being compared with a COMPUTATIONAL or COMPUTATIONAL-1 field, *or* a COMPUTATIONAL or COMPUTATIONAL-1 field is being compared with a DISPLAY field which is greater than 12 digits, *or* both operands are literals or the statement contains an operand longer than 120 characters, or one of the fields is a non quasi-character group field.	Code is generated to indicate that the first operand of the comparison is greater than the second.
313 GROUPED FIELD EXCEEDS 120 CHARACTERS	The literal is an ALL literal and the field is greater than 120 characters.	Only 120 characters are taken.
314 ILLEGAL MOVE: EDITED NUMERIC TO NUMERIC/ ALPHABETIC	It is not possible to de-edit a field. Note: There is no error message if the target field is alphanumeric.	The source field is assumed to be unedited alphanumeric.
315 MOVE: SOURCE TOO BIG FOR ARITHMETIC CONVERSION	The field needs conversion but exceeds the arithmetic size limit.	No code is generated for the MOVE statement.
316 (unused)		
317 ILLEGAL GROUP OR RECORD MOVE	If the source or result field is a group or record, both must start on a character boundary and be a whole number of characters in length. If not, they must both be the same size, and if the source field is subscripted each member of the table must start on the same bit within a word, and if the target field is subscripted, each member of that table too must start on the same bit within a word.	No code is generated for the MOVE statement.
318 ILLEGAL MOVE: NUMERIC TO ALPHABETIC	The fields are incompatible.	Code is generated assuming the target field to be alphanumeric.
319 VALUE CLAUSE ILLEGAL	*Either* the field is edited, *or* it is a group or record and either does not start on a character boundary or is not a whole number of characters in length.	No value is loaded in the object program.
320 ILLEGAL MOVE: ALPHABETIC TO NUMERIC	As printed.	Code is generated assuming the source field to be alphanumeric.

Message	Cause	Compiler action
321 LITERAL NEEDS CONVERSION AND EXCEEDS SIZE LIMIT	A numeric literal is associated (in a MOVE, VALUE or IF statement) with a COMPUTATIONAL-1 or sterling field, and hence requires conversion.	
	Either	
	(a) The field is sterling and the number of digits before the decimal point exceeds 12,	(a) Truncates excess digits to the left,
	or	
	(b) the field is sterling and the number of digits after the decimal point in the literal exceeds 14,	(b) accepts only the first 14 digits,
	or	
	(c) the field is COMPUTATIONAL-1 and the number of digits before the decimal point in the literal exceeds 12,	(c) assumes a value of zero,
	or	
	(d) the field is COMPUTATIONAL-1 and the number of bits in the literal (= decimal size × 3.322 rounded up) *plus* the number of bits after the point in the field, exceeds 46.	(d) assumes a value of zero.
322 /DECIMAL POINT/> 12	All these messages indicate that one of the limits involved in converting one type of field to another has been exceeded. They may arise from such statements as: (a) MOVE (non-COMPUTATIONAL-1 field) TO (COMPUTATIONAL-1 field) (b) MOVE (COMPUTATIONAL-1 field) TO (non-COMPUTATIONAL-1 field) (c) arithmetic statements where at least one of the source fields is different from the target field and one of the two differing fields is COMPUTATIONAL-1.	322 Contents are assumed to be already converted, i.e. no conversion code is generated.
323 /BINARY POINT/> 46	All these involve conversions from decimal to binary, or vice versa. The general rules for conversion (by the compiler) are:	323 As above.

Message	Cause	Compiler action
	Decimal to binary: multiply by 2^n where n is the number of binary point positions in the target field, and *divide* by 2^m where 2^m is the nearest approximation (rounded upwards) to 10^p and p is the number of decimal point positions in the source field. *Binary to decimal:* the reverse of the above (i.e. divide instead of multiply, and vice versa). The effect of this can be seen from the following examples involving MOVEs, where the fields have the quoted PICTUREs: 1 9999V99 to 1(13)V1(5): multiply by 2^5 (shift left 5 places) and divide by '100' (2^7). 2 9(7)P to 1(32)V1(6): multiply by 2^6 and divide by '1/10' (i.e. multiply by '10'). 3 1(36)V1(7) to 9(8)V999: multiply by '1000' and divide by 2^7. 4 1(28)P(5) to 9(7)PP: multiply by 2^5 (=divide by $1/2^5$) and divide by '100'. (To maintain accuracy, divisions are always done after multiplications). The effective resultant size can now be determined. For example, in 3 above, we multiplied by '1000', equivalent to shifting left 10 places (since 2^{10} = 1024), and then shifted right 7 places. The resultant increase is 3 bits, and so the new size of the converted source value will be 43 + 3 bits = 46 bits. If the new size is negative, there is no 'overlap'.	
324 NO OVERLAP ON CONVERSION		324 Code is generated to make the converted result zero.
325 OUT OF RANGE ON CONVERSION		325 The size of source field is reduced until the result will be in range, and code is generated to convert on the basis of this size.

Message	Cause	Compiler action
	If the new size is > 46, (COMPUTATIONAL-1 fields) or > 40 (COMPUTATIONAL fields: = approximately 10^{12}) then the result is 'out of range'. The compiler assumes that it is in range by reducing the size of the source, i.e. it would assume the source for 3 above as 1(30)V1(7). If the reduction results in the source being of negative length, then again there is no overlap.	
326 COULD EXCEED ARITHMETIC LIMITS FOR STERLING	(a) There will be an overflow when multiplying in order to round a sterling target field. e.g. a source field with PICTURE 9(6)P(7) USAGE DISPLAY holds pence which are to be converted and stored in a sterling target field with PICTURE 9(6)P. If rounding is specified, it it clear that the rounding factor is 6^d. To add this to the source field necessitates multiplying	(a) The rounding clause is ignored. (b) The results may be unpredictable.

Message	Cause	Compiler action
	the contents of the source field by 10^7. This causes overflow since the arithmetic size limit of 12 digits will be exceeded. (b) The result of an arithmetic statement needs multiplying to align it with the sterling target field.	
327 TOO BIG ON CONVERSION, TAKE ZERO	Explanation as for 326 above.	Code is generated to make the converted result zero.
328 (unused)		
329 (unused)		
330 (unused)		
331 (unused)		
332 88-LEVEL ITEM HAS TOO MANY VALUES	The compiler limit is exceeded.	The statement is ignored.
333 (unused)		
334 VALUE CONSTANT TOO LARGE	An 88-level item is larger than the field it refers to.	A value of zero is assumed.
335 to 339 (unused)		
340 MORE THAN ONE RECORD READ INTO FILE	As printed.	The statement is ignored.
341 to 399 (unused)		
400 TOO FEW ACTUAL SUBSCRIPTS	There are fewer subscripts in the procedure statement than required by the definition of the table in the Data Division (and see comments on message 107).	The first subscript is accepted and the missing ones are given the value 1.
401 TOO MANY ACTUAL SUBSCRIPTS	There are more subscripts in the statement than required by the definition of the table.	The numbers required are taken from the beginning of those given and the rest are ignored.
402 INVALID SUBSCRIPT	A data name subscript is non-elementary.	The value 1 is assumed for the subscript.
403 INVALID SUBSCRIPT	A data name subscript is alphabetic, sterling or edited.	The value 1 is assumed for the subscript.
404 INVALID SUBSCRIPT	A numeric data name subscript does not have its point position immediately after the least significant digit.	The contents of the field are assumed to be integral i.e. 1.23 is taken as 123 and 1200 in a POINT RIGHT 2 field is taken as 12.
405 SUBSCRIPT INVALID	A data name subscript is described in the Data Division as itself OCCURring.	The first occurrence of the data name is accepted.

Message	Cause	Compiler action
408 COMPOSITE TOO BIG	When printed as a CAUTION: the maximum possible result exceeds 12 digits (decimal) or 46 bits (COMPUTATIONAL-1). When printed as an ERROR: the intermediate result of a MULTIPLY with both decimal and COMPUTATIONAL-1 operands has exceeded 92 bits.	When printed as a CAUTION: code is generated assuming that the intermediate and final result will not exceed these limits, but in the case of DIVIDEs the message will always mean loss of accuracy. When printed as an ERROR: the statement is ignored, i.e. the result field will be unchanged.
409 ENTER: LITERAL HAS IRRELEVANT POINT	Literal parameters are passed on in character form. It is assumed that the ENTERed routine knows the position of the point.	The point is omitted and the gap is closed up.
410 NON-NUMERIC RESULT FIELD	The result field for an ADD/SUBTRACT... GIVING statement is alphanumeric.	Code is generated assuming the result field to be numeric.
411 (unused)		
412 ILLEGAL OPERAND	(a) the result field of an ADD/SUBTRACT... GIVING statement is alphabetic or non-elementary, *or* (b) the result field of a MULTIPLY/DIVIDE statement is alphabetic, or alphanumeric, or non-elementary, *or* (c) in a MULTIPLY or DIVIDE statement the multiplicand or dividend or divisor is alphabetic or alphanumeric or non-elementary, or edited, or exceeds the maximum size for an arithmetic operand, *or* (d) any other operand of an ADD/SUBTRACT statement except the result field of a 'GIVING' is non-elementary or edited, or alphabetic, or alphanumeric, or exceeds maximum size for an arithmetic operand.	In the case of the first three causes, (a) to (c), the statement is ignored, i.e. the contents of the result field will be unchanged. In the case of cause (d) the operand in error is ignored: this may lead to the result field being zeroized if every operand is wrong.
	(e) in an ADD/SUBTRACT CORRESPONDING statement a pair of items correspond in all respects except that the class of one or both of them is not numeric. (The message in this case may be ignored)	In the case of cause (e) no code is generated for this pair of items (as the programmer would have expected)

Message	Cause	Compiler action
413 TOO MANY OPERANDS	As printed: at present the maximum permitted number of operands in an arithmetic statement is 9. See also page 268.	Operands after the ninth are ignored.
414 to 435 (unused)		
436 TOO MANY OPERANDS	A statement containing subscripts has too many operands to be processed at one time.	Generation of code ceases.
437 to 499 (unused)		
500 GO DEPENDING DATA NAME INVALID	The depending field is non-elementary, non-integral, or alphabetic, or a literal.	The statement being processed is ignored.
501 (unused)		
502 (unused)		
503 (unused)		
504 (unused)		
505 (unused)		
506 SECTION EXCEEDS 32767 WORDS	As printed: if the source program is not divided into sections, the message refers to the whole program. This condition is checked at the beginning of each new paragraph or section. See also page 268.	Generation of code ceases, and the next program map line starts at address '2'. The program should be segmented or re-segmented.
507 ALTER OPERAND HAS ILLEGAL PRIORITY NO	The operand of the ALTER statement is in an overlay segment of priority > 49, but the ALTER statement itself is not in the same overlay segment.	The illegal part of the ALTER statement is ignored.
508 TOO MANY LABEL REFERENCES	A cue table of 'forward references' made up and used by the compiler is full. See also page 269	Generation of code ceases.
509 (unused)		
510 SORT PROCEDURE NOT A SECTION	as stated	Invalid object coding will be generated
511 TOO MANY SORT PROCEDURE NAMES	The Source program contains >12 input/output procedures.	SORT statements containing the excess procedures will generate invalid coding
512 to 566 (unused)		
567 INVALID DATA DECL.	Part of the data declaration sentence is invalid.	The data name and its level number are processed: the rest of the sentence is ignored.
568 SUPPRESS ALONE	SUPPRESS met without preceding ZERO.	The sentence being processed is ignored.
569 SECTION NAME MISSING	SECTION is met without a valid preceding name.	The SECTION header is ignored.

Message	Cause	Compiler action
570 ELSE ALONE	ELSE or OTHERWISE out of context.	The sentence being processed is ignored.
571 UNTIL MISSING	UNTIL not found before FIRST in an EXAMINE statement.	The statement is processed assuming the presence of UNTIL.
572 UPON ALONE	UPON is met out of context.	The statement being processed is ignored.
573 to 577 (unused)		
578 FILE ALONE	Keyword FILE is out of context.	The statement being processed is ignored.
579 REEL MISSING	Keyword MULTIPLE met without REEL.	The statement is ignored (as it would be even with the presence of REEL).
580 NO SENTENCE	SENTENCE missing after NEXT.	The presence of SENTENCE is assumed.
581 NEXT MISSING	SENTENCE met without preceding NEXT.	The statement being processed is ignored.
582 FILLER MISUSED	FILLER out of context.	The statement being processed is ignored.
583 FD MISUSED	FD out of context.	The statement being processed is ignored.
584 PERIOD MISSING	The full stop is missing after a paragraph or section declaration.	The presence of the full stop is assumed. (c)
585 OPTION NOT IMPLEMENTED	The sentence contain a COBOL option not implemented in this version of the compiler.	The statement containing the option is ignored.
586 DUPL HARDWARE/ CONDITION NAME	A hardware or condition name is not unique.	The compiler will take the last occurrence of the duplicated name in the Data Division: this may be a data name, or the wrong occurrence, so causing further error messages and/or invalid coding.
587 NO ENTER	The keyword USING is out of context.	The statement being processed is ignored.
588 MISSING THRU	The word THRU is missing in a RANGE statement.	No effect on compilation.
589 OF MISUSED	The keyword OF is out of context.	The statement being processed is ignored.
590 MISSING SUPPRESS	ZERO SUPPRESS statement has invalid format.	The statement being processed is ignored.
591 LONE OFF	The keyword OFF is out of context.	The statement being processed is ignored.
592 ON ALONE	The keyword ON is out of context.	The statement being processed is ignored.
593 DEPENDING ALONE	The keyword DEPENDING is out of context.	The statement being processed is ignored.
594 ELSE/OTHERWISE ALONE	ELSE or OTHERWISE out of context.	The statement being processed is ignored.

Message	Cause	Compiler action
595 BIT-N CLAUSE INVALID	As printed.	The clause is ignored.
596 REEL ALONE	REEL out of context.	The statement being processed is ignored.
597 MISSING IS	IS missing in a BIT-N clause.	The statement being processed is ignored.
598 INVALID RERUN	Invalid RERUN clause.	The clause is ignored.
599 INVALID SELECT	Invalid SELECT... ASSIGN... statement.	The statement is ignored.
600 to 699 (see pages 201 to 204)		
700 COMPILER STORAGE EXCEEDED: PICTURE IGNORED	Previous field description is too long/complicated. See also page 270.	The current picture and elementary clauses are ignored.
701 ILLEGAL MODE	(a) Card reader: RECORDING MODE specified is not equal to 2 or 0. (b) Any other peripheral: a non-zero RECORDING MODE is specified.	In all cases the mode is assumed to be zero.
702 FILE INFORMATION CLASH	Magnetic tape files: *either* (a) LABEL RECORDS OMITTED specified, *or* (b) Both BLOCK-ROCORDS and BLOCK-CHARACTERS specified, *or* (c) VALUE OF ID clause missing, *or* (d) The file-identity is larger than 12 characters, *or* (e) TYPE clause specified for a cassette tape. Slow peripheral files: various clauses (e.g. VALUE OF ID, LABEL RECORDS) have been used which apply only to magnetic tape or other fast peripheral files.	(a) The clause is ignored. (b) The BLOCK-CHARACTERS clause is ignored. (c) A file identity SCRATCH -TAPE is assumed. (d) The first 12 characters are taken as the identity. (e) The clause is ignored. Slow peripheral files: the clauses are ignored. (c)
703 TAPE MAY BE LOST ON CLOSING	Magnetic tape files only; either the clause ACTIVE-TIME is missing or the value specified is zero.	No compiler action. (c)
704 BLOCK SIZE WRONG	(a) Any peripheral: the size of the largest record in the file exceeds maximum permitted (e.g. 80 characters for card output file). (b) Magnetic tape: size of record is greater than that specified in BLOCK -CHARACTERS clause. (c) Size specified in a BLOCK-CHARACTERS clause does not conform to the rules for EDS MCF or FDS files.	(a) No corrective action i.e. the user-defined size is accepted. (b) The BLOCK-CHARACTERS clause is ignored. (c) The size is rounded up to the next acceptable figure.

Message	Cause	Compiler action
705 INVALID RETENTION PERIOD	The literal in the ACTIVE -TIME clause contains a non-numeric character.	A retention period of 30 is assumed.
706 FILE UNASSIGNED	SELECT...ASSIGN statement is missing in Environment Division for a file defined in the Data Division, or a sort-file has been incorrectly assigned.	The file is assigned to Magnetic Tape Unit 0
707 ILLEGAL OCCURS	*Either* the field being defined as OCCURring has a level <02 or >49, *or* the literal in the OCCURS clause is zero or contains a non-numeric character, *or* a table has more than 3 levels, *or* a field occurs more than 262,143 times, *or* a table as defined would require more than 349,525 words of storage.	In all cases the current OCCURS statement is ignored.
708 ILLEGAL REDEFINES	*Either* a 01 level in the File Section is being redefined, *or* a redefining field or a subfield of the redefining field has an initial value, *or* the redefined and redefining fields have different levels, *or* they are not consecutive at their level, *or* they have different sizes (at levels other than 01).	In the first four cases the redefining clause is ignored. In the fifth case store is allocated equivalent to the larger of the two fields. The message will also be printed if a redefining record (01 level) is larger than that redefined and all lower allocation has been used. The redefining record will then go into Upper and any part of the user's program that expects the two records to share the same area will be invalid.
709 ILLEGAL RENAMES	Either the renaming item is not a 66 level, or the level of the renamed item is not in the range 2 to 49, incl., or the renamed item is not in the immediately preceding record, or the renamed item is subscripted, or the second item in the RENAMES... THRU... clause is a subfield of the first.	In all cases the RENAMES clause is ignored.

Message	Cause	Compiler action
710 LEVEL ERROR	Either (a) the first field after a section heading or file description has a level in the range 2 to 49 (inclusive), or	(a) 01 is assumed.
	(b) the level is in the range 2 to 49 (incl.) and between the last 01 level and the current level there is a level number greater than the current level, but no level equal to current level, or	(b) No compiler action.
	(c) a 66 level is followed by non-66 levels in the same record, or	(c) Arbitrary field description.
	(d) an FD level has been met in the File Section and the first line in the section was not itself an FD level, or was an FD level entry rejected because of syntax errors.	(d) Compilation halts, displaying SERVICED ..... ABANDON.
711 SUBFIELD USAGE INCOMPATIBLE	As printed.	No corrective action. (c)
712 USAGE ERROR	More than one USAGE clause specified for a field.	Second and subsequent USAGE clauses are ignored.

Message	Cause	Compiler action
713 ILLEGAL VALUE	A value has been assigned either (a) to a subfield of a major field which has been given a value, or (b) to an occurring field or a subfield of an occurring field, or (c) to a redefining field or a subfield of a redefining field, or (d) to a field whose level is not 77, 88 or in the range 1 to 49 (inclusive), or (e) to a field in the LINKAGE SECTION.	(a) Both values are accepted. In all other cases the value is ignored.
714 NOT ENOUGH LOWER	See page 270.	
715 CLASS ERROR	More than one CLASS clause specified for a field.	Second and subsequent CLASS clauses are ignored.
716 SUBFIELD CLASS INCOMPATIBLE	As printed.	No corrective action. (c)
717 OBJECT COMPUTER MEMORY INVALID/ MISSING	*Either* no OBJECT-COMPUTER MEMORY... WORDS clause in the Environment Division *or* the memory size specified is out of range.	The memory size is taken as 8K.
718 OBJECT COMPUTER NAME INVALID/ MISSING	*Either* no OBJECT-COMPUTER name is given *or* the name is invalid.	1902 is assumed
719 SEGMENT-LIMIT INVALID	The segment limit is not a legal integer, *or* = 0, *or* is >49.	A segment limit of 49 is assumed.
720 ILLEGAL TRIGGER PERIPHERAL	RERUN statement contains END OF REEL clause referring to a slow peripheral file.	No corrective action.
721 TRIGGER FILE UNASSIGNED	The file causing a dump has not been defined in a SELECT... ASSIGN statement.	The RERUN statement is ignored.
722 NUMBER OF RECORDS ILLEGAL	The number of records causing a dump = 0, *or* >8,000,000 *or* contains a non-numeric character.	The number is assumed to be 10,000
723 SECOND DUMP IGNORED	More than one RERUN statement has been specified.	All but the first statement are ignored.
724 ILLEGAL DUMP PERIPHERAL	The dump file specified is not a magnetic tape file.	The dump file is assumed to be TAPE-*.
725 DUMP FILE UNASSIGNED	The dump file has not been defined in a SELECT... ASSIGN statement.	The dump file is assumed to be TAPE-*.
726 FILE ASSIGNED TWICE	More than one SELECT... ASSIGN statement for a file.	Only the first statement is considered.
727 ILLEGAL DEVICE NUMBER	The hardware device number specified = 0, *or* >63.	Number is assumed to be 0.

Message	Cause	Compiler action
728 ILLEGAL PROPERTY CODE	The literal specifying the TYPE of hardware device contains a character > 8.	The TYPE is assumed to be 0.
729 TOO MANY BUFFERS	The number specified in a RESERVE ALTERNATE AREAS clause is > 1.	One ALTERNATE AREA is assumed. (c)
730 FILE-NAME UNDEFINED	A file specified in a SAME AREA or SAME RECORD AREA statement has not been defined by a SELECT... ASSIGN statement.	The file name is ignored.
731 ILLEGAL SPECIAL NAME	A fast peripheral is specified in a Special-Names statement.	The statement is ignored.
732 INCONSISTENT TYPES	Different TYPES have been specified for the same peripheral.	No corrective action – both TYPES are accepted. (c)
733 (unused)		
734 (heading or clause) MISPLACED	One of the clauses or headings which are required to precede the current clause/heading has not been met. (Environment Division clauses.)	The missing clause is assumed.
735 (heading or clause) MISPLACED	One of the previous clauses/ headings should only appear after the current clause or heading. (Environment Division clauses.)	No compiler action.
736 NEEDS FILLER (A)	The record as it stands would lead to incorrect data allocation (e.g. character field not starting on character boundary).	The record is adjusted by insertion of fillers. (c)
737 NEEDS FILLER (S)	As 736, referring to an occurring field.	As above. (c)
738 COMPILER STORAGE EXCEEDED	*Either* (a) there are too many files in the source program, *or* (b) there are too many and/or complex fields in a record. See also page 270.	(a) The compilation is abandoned. (b) All fields up to the end of the current record are given an arbitrary field description.
739 TOO MANY FILES	There are too many files in the source program (compiler limit undefined). See also page 270.	All references to files in the Procedure Division will generate incorrect object coding.
740 ELEMENTARY SIZE S1(24) CHANGED TO S1(23)	There is no gain in arithmetic length from specifying S1(24).	The compiler changes the definition, as printed.
741 INVALID PICTURE	The letter in the message will be one of the following, with the given interpretation:	
	A – illegal character.	A – the illegal character is ignored.
	C – misplaced report sign or illegal combination of characters (e.g. DR instead of CR).	C – the illegal character is ignored.

Message	Cause	Compiler action
	D – Incompatible characters (e.g. £1). E – the physical size of the PICTURE is > 30 characters. H – FLOAT/SUPPRESS error (e.g. + + + Z). J – bracket error.	D – the second character is ignored. E – no action (does not affect compilation). H – the invalid float characters (e.g. Z in the example) are assumed to be '9's. J – bracket is ignored.
742 NON-NUMERIC CHARACTER IN LITERAL	As printed	A value of 1 is assumed.
743 RECORD < 4 CHARS	The largest record in a file is < 4 characters in length.	No corrective action
744 ELEMENTARY CLAUSE IN GROUPED FIELD	A clause applicable only to an elementary field (e.g. SIGNED) has been specified for a major field, or there is more than one PICTURE.	The clause is ignored.
745 NO SIZE/PICTURE	Neither a SIZE nor a PICTURE clause has been specified for an elementary field, or there is more than one PICTURE.	A size of 1 is assumed.
746 ILLEGAL ELEMENTARY SIZE	The object program size of an elementary field is *either* >120 characters, *or* >43 bits, for a COMPUTATIONAL field, *or* >48 bits, for a COMPUTATIONAL-1 field *or* =0.	In all cases a NUMERIC DISPLAY field of 1 character is assumed.
747 CLASS AND USAGE INCOMPATIBLE	As printed, e.g. CLASS AN, USAGE DISPLAY-3.	No corrective action.
748 ILLEGAL CLASS/USAGE	The CLASS or USAGE specified is incompatible with other clauses specified for the field.	A CLASS and USAGE are assumed compatible with the other clauses specified.
749 ILLEGAL FLOAT/ SUPPRESS	*Either* (a) more than one of the clauses ZERO SUPPRESS, CHEQUE PROTECT or FLOAT SIGN has been specified for a field, *or* (b) one of the above clauses has been specified for a SIGNED field, *or* (c) the effect of the LEAVING option as specified implies that no character would be suppressed.	(a), (b) No corrective action. (c) The LEAVING option is ignored.
750 INVALID BLANK WHEN ZERO	BLANK WHEN ZERO has been specified for a field which is also described as *either* COMPUTATIONAL-1 *or* ALPHABETIC *or* whose PICTURE contains A,X,1 or *.	In all cases the BLANK WHEN ZERO clause is ignored.
751 TOO MANY INSERTIONS	More than 15 insertion characters have been specified in the PICTURE.	A NUMERIC DISPLAY field of 1 character is assumed.

Message	Cause	Compiler action
752 POINT OUT OF RANGE	The point position specified would require the size of the field to be *either* >12 character *or* >48 bits for a COMPUTATIONAL-1 field.	No corrective action. (c)
753 ILLEGAL STERLING SIGN	The sign of a sterling field coincides with the 10d position.	No corrective action.
754 (unused)		
755 ILLEGAL 77 LEVEL	*Either* the 77 level is in the FILE SECTION *or* the 77 level has a subfield.	No corrective action. (c)
756 ILLEGAL 88 LEVEL	*Either* (a) a condition name (88 level) does not have a VALUE clause, *or* (b) there are more than 18 values associated with a condition name.	VALUE 9 is assumed.
757 MAJOR FIELD HAS VALUE: SUBFIELD SYNCH./NON -CHARACTER	As printed.	No corrective action. (c)
758 INCOMPATIBLE CLAUSES	The clauses specified cannot occur in the same field description, e.g. both SYNC LEFT and SYNC RIGHT specified for a field.	The clause which appears first is accepted, the following clause which produces the incompatibility is ignored.
759 SORT INPUT FILE INVALID	The input file is not on an allowable sort medium, or is not on the same device type as the sort-file	The SORT statement is ignored
760 SORT OUTPUT FILE INVALID	As above, but for the output file.	The SORT statement is ignored.
761 SORT KEY OCCURS	A key either contains an OCCURS clause or is subordinate to a field that contains an OCCURS clause	The first occurrence is accepted
762 INVALID SORT KEY	A key is a group field and is not an integral no. of characters in length	The key is ignored
763 MORE THAN 10 SORT KEYS	As stated	The extra keys are ignored
764 SORT KEY IS STERLING	As stated	The key is treated as alphanumeric (C)
765 SORT-* NOT WORD ADDRESS	The data-name in the SORT-* clause does not begin on a word boundary	Code is generated to address the word containing the start of the field
766 RELEASE NOT FOR SORT FILE	As stated	The RELEASE statement is ignored.
767 MORE THAN ONE RECORD IN RETURN INTO FILE	The relevant sort-file has more than one record description	A record equivalent to the size of the first record in the file is moved into the target area
768 to 799 (unused)		

Message	Cause	Compiler action
800 PRIORITY INVALID/ MISSING – 01 ASSUMED	As printed – refers to the last two digits of the PROGRAM-ID.	The priority is assumed to be 01.
801 DUPL. SPECIAL NAME - dd...	An operand in a procedural statement can be either a 'special name' or a data name.	The operand is taken to be the data name. dd... is the name, of not more than 30 characters.
802 INVLD. QUALIFYING - dd...	Invalid qualification (see rules on page 17).	The last occurrence of the data name (dd...) in the Data Division is taken as the one referenced.
803 NO FILE	FILE is missing from an ACCEPT FILE KEYS ... statement.	The presence of FILE is assumed. (c)
804 EXAMINE MISSING	TALLYING or REPLACING met without preceding EXAMINE.	The statement being processed is ignored.
805 LEVEL NUMBER MISSING	As printed – refers to a data field declaration.	The level number is assumed to be the same as that of the preceding data field.
806 NO KEYS	KEYS missing after ACCEPT FILE.	The presence of KEYS is assumed. (c)
807 NO BY	BY missing in a PERFORM... VARYING... [AFTER...] statement.	The presence of BY is assumed. (c)
808 INVALID QUALIFIER(S)	The data name qualifier(s) following IN or OF are missing or invalid.	The invalid qualifiers are ignored: an attempt is made to identify the qualified field using the operands up to the point of error.

Message	Cause	Compiler action
809 LONE EXAMINE	EXAMINE met without TALLYING or REPLACING after it.	The statement being processed is ignored.
810 BEFORE ALONE	BEFORE is out of context.	The statement being processed is ignored.
811 IN MISUSED	IN is out of context.	The statement being processed is ignored.
812 INVALID DISPLAY PERIPHERAL	The peripheral referenced in a DISPLAY... UPON... statement, as defined in the Special Names Section, is not a card punch, paper tape reader or printer.	No object coding is generated for the statement.
813 PERFORM TIMES LITERAL ILLEGAL	The literal in a PERFORM ... TIMES statement is negative or contains a decimal point.	In the first case the modulus is taken, in the second the decimal point is ignored.
814 ILLEGAL LEVEL NUMBER	Either (a) The variable in a PERFORM ...VARYING statement is a file or (b) A condition-name has initially been interpreted as part of an abbreviated condition.	(a) Invalid coding will be produced. (b) The condition-name will be correctly interpreted as such.
815 ACCEPT PERIPHERAL NOT VALID	The peripheral referenced in an ACCEPT ... FROM ... statement, as defined in the Special Names Section, is not a paper tape reader or card reader.	No object coding is generated for the statement.
816 COMPILER LIMIT EXCEEDED	A procedural sentence is too long and/or complex: e.g. an ADD statement with many operands, some being long literals, or an IF statement involving a condition name with complex multiple VALUEs.	The statement being processed is ignored. The programmer must break the statement down into smaller, less complex elements.
817 PERFORM TIMES INVALID	The ... TIMES variable, if a data name, is: non-elementary, *or* non-numeric, *or* numeric edited, *or* non-integral, *or* exceeds the arithmetic size limit or if a literal, it is non-numeric, *or* non-integral, *or* negative, *or* greater than 8,388,607.	The 'TIMES' part of the statement is ignored, i.e. the PERFORM will be performed once only.
818 MORE THAN 1 RECORD IN READ INTO FILE	All the records in READ ... INTO ... statements must be of same size.	A record equivalent to the size of the *first* record in the file is moved into the target area.
819 CLOSE REEL/LOCK INVALID	This option is not yet implemented.	CLOSE ... REEL is assumed.

Message	Cause	Compiler action
820 CLOSE REEL NOT ON MAGNETIC TAPE	As stated.	CLOSE of non-magnetic tape file is assumed.
821 EXAMINE LITERAL NOT 1 CHARACTER	As stated.	The first character only is taken.
822 EXAMINE FIELD NON -CHARACTER	The field to be EXAMINEd is not a character field.	The statement is ignored.
823 EXAMINE LITERAL INVALID FOR FIELD	As stated, e.g. an alpha character is quoted for a numeric field (see page 86).	The statement is accepted as if valid.
324 EXAMINE DATA-NAME IS A FILE	As stated.	The EXAMINE statement is ignored.
825 UNALTERED INCOMPLETE GO	The GO statement has a blank address and is not the subject of an ALTER statement.	Code is produced to call the run-time subroutine %BLANK which types the message (at run-time) 'ATTEMPT TO OBEY AN UNALTERED GO SENT nnnnnn' where nnnnnn is the sequence number of the GO statement.
826 NO SAME	The keyword SAME is missing before AREA or RECORD.	AREA or RECORD is ignored.
827 (unused)		
828 PROG-ID INCORRECT	*Either* (a) the PROGRAM-ID statement is missing or different to the name quoted in the *IDENTITY steering line, *or* (b) the name quoted in the *IDENTITY steering line for a master program contains more than 4 characters, *or* (c) the name quoted in the *IDENTITY steering line for a subroutine contains more than 11 characters.	(a) The name quoted in the *IDENTITY steering line is taken. If there is no such steering line the name COBOL is assumed. (b) Only the first 4 characters are taken. (c) The first 11 characters are taken.
829 DUPL. NAME dd..nn	The same name (and qualification structure, if any) has been used to define more than one field in the Data Division.	The *last* occurrence of the name is assumed to be the one referenced. dd... is the name of not more than 30 characters, nn is the number of duplications.
830 TIME DATANAME NOT ALPHANUMERIC	The field into which TIME is to be accepted is not alphanumeric.	If the field is a character field the statement is processed; if it is not a character field the statement is ignored.
831 LEFT/RIGHT ALONE	LEFT or RIGHT out of context.	The statement being processed is ignored.
832 SIGN ALONE	SIGN out of context.	The statement being processed is ignored.
833 JUSTIFIED ALONE	JUSTIFIED out of context.	The statement being processed is ignored.
834 POINT ALONE	POINT out of context.	The statement being processed is ignored.

Message	Cause	Compiler action
835 NO SIGN	SIGN missing after FLOAT.	The statement being processed is ignored.
836 CONDITION FIELD INVALID	The field with which a condition name (88 level) is associated is a group field which is *either* >120 characters long *or* is not a quasi-character field.	The statement being processed is ignored.
837 NO DATA NAME	Name missing/invalid in data declaration statement.	The word is ignored.
838 88-LEVEL ITEM HAS TOO MANY VALUES	As printed (variable compiler limit).	The statement being processed is ignored.
839 VALUE CONSTANT TOO LARGE	An 88 level item is larger than the field it refers to.	A value of zero is assumed.
840 LONE VARYING	VARYING out of context.	The statement being processed is ignored.
841 (unused)		
842 DIGITS ALONE	DIGITS out of context.	The statement being processed is ignored.
843 (unused)		
844 (unused)		
845 BITS ALONE	BITS out of context.	The statement being processed is ignored.
846 FILE ALONE	FILE is encountered with neither SECTION nor RECORDS following it.	The statement being processed is ignored.
847 CHARACTERS ALONE	CHARACTERS out of context.	The statement being processed is ignored.
848 FIRST MISSING	FIRST not found after UNTIL in an EXAMINE statement.	The statement is processed assuming the presence of FIRST.
849 KEYS ALONE	KEYS out of context.	The statement being processed is ignored.
850 LONE PROGRAM	PROGRAM met without preceding EXIT.	The statement being processed is ignored.
900 SEEK STATEMENT NOT FOR FILE	The operand of the SEEK statement is not a file name.	The statement is ignored.
901 SEEK NOT FOR MASS STORAGE FILE	The file name in the SEEK statement does not refer to a file on a direct.access device.	The statement is ignored.
902 SEEK FOR FILE ACCESSED SEQUENTIAL	The file name in the SEEK statement has been defined as ACCESS SEQUENTIAL.	The statement is ignored.
903 NON-RECORD DELETED	The operand of a DELETE statement is not a record in FILE SECTION.	The statement is ignored.
904 INVALID KEY NOT ON EDS/MCF/FDS	The READ statement being processed has an INVALID KEY clause and yet the file is not on a direct access file.	AT END is assumed in place of INVALID KEY.

Message	Cause	Compiler action
905 INVALID KEY FOR SEQUENTIAL FILE	The statement being processed has an INVALID KEY clause and the file is not specified as ACCESS RANDOM.	If the statement is a READ the clause is treated as AT END, otherwise it is ignored.
906 AT END FOR RANDOM FILE	The statement being processed has an AT END clause and the file is specified as ORGANIZATION DIRECT.	The clause is treated as INVALID KEY.
907 DELETE NOT FOR MASS STORAGE FILE	The record name in the DELETE statement does not belong to a file assigned to a direct access device.	The statement is ignored.
908 INVALID KEY CLAUSE MISSING	The file to which the statement refers has been defined as ORGANIZATION DIRECT and the statement has no INVALID KEY clause.	Code is generated to call a standard routine which will type an error message (see EXCEPTION TYPE 02 on page 250) should an invalid key condition occur when the program is run.
909 REWRITE NOT FOR MASS STORAGE FILE	The record-name in the REWRITE statement does not belong to a file assigned to a direct access device.	The statement is ignored.
910 SEQUENTIAL WITH DIRECT	ACCESS SEQUENTIAL and ORGANIZATION DIRECT are specified for the same file.	Reference to this file may generate incorrect coding.
911 SEQUENTIAL WITH SYMBOLIC	SYMBOLIC KEY is specified for a file for which ACCESS SEQUENTIAL is stated.	The SYMBOLIC KEY clause is ignored.
912 SEQUENTIAL WITH ACTUAL	ACTUAL KEY is specified for a file for which ACCESS SEQUENTIAL is stated.	The ACTUAL KEY clause is ignored.
913 RANDOM WITHOUT SYMBOLIC	No SYMBOLIC KEY has been specified for a file with ACCESS RANDOM.	References to this file may generate incorrect object coding.
914 INDEXED WITH ACTUAL	ACTUAL KEY is specified for a file with ORGANIZATION INDEXED.	The ACTUAL KEY clause is ignored.
915 DIRECT WITHOUT ACTUAL	No ACTUAL KEY has been specified for a file with ORGANIZATION DIRECT.	References to this file may generate incorrect object code.
916 REPEATED FILE INF. - IGNORED	One of ACCESS MODE, ORGANIZATION, ACTUAL KEY or SYMBOLIC KEY appears more than once under a SELECT statement.	All but the first of each type is ignored.
917 ILLEGAL F-TYPE IN APPLY	APPLY TRANSFER-REPLY to a file which is not on E.D.S. M.C.F., F.D.S. or printer. APPLY FILE-MESSAGES to a file which is not slow peripheral.	The APPLY information is ignored.

Message	Cause	Compiler action
918 NAME UNDEFINED :- aaa --- aaa	The name in the message has been used in an ACTUAL KEY, APPLY, or similar clause describing a file and has not been defined in the DATA DIVISION or has an illegal level (i.e. FD or 88).	The data-name TALLY is assumed.
919 ACTIVE-TIME DATA NAME INVALID	See under The FD statement.	
920 OPEN INVALID FOR THIS DEVICE	OPEN I-O has been specified for a file that is not on a direct access device; INPUT for an output device; and similar.	The statement is ignored.
921 INVALID OPEN AT END (ACCESS RANDOM)	OPEN OUTPUT ...AT END has been specified for a file with ACCESS RANDOM.	The statement is ignored
922 SYMBOLIC KEY INVALID	The field defined as SYMBOLIC KEY for the file referred to in the statement is not a character field or exceeds 64 characters.	No code generated for *processing the SYMBOLIC KEY only.*
923 ACTUAL KEY INVALID	The field defined as ACTUAL KEY for the file referred to in the statement is not numeric or exceeds 6 digits.	No code is generated for *processing the ACTUAL KEY only.*
924 VALUE OF ID DATANAME INVALID	See under The FD statement.	
925 GENERATION-NO DATA NAME INVALID	See under the FD statement	
926 INDEX BUFFER NOT WORD ADDRESS	The field defined for INDEX-BUFFER is not a multiple of 24 bits long and/or does not commence on a word boundary.	Code is generated to address the word containing the start of the field.
927 FILE INF. INCOMPATIBLE	Direct access clauses appear in a non-DA SELECT .... ASSIGN statement.	The clauses are ignored.
928 NO ACCESS MODE	There is no ACCESS MODE clause for a file assigned to a direct access device.	ACCESS SEQUENTIAL is assumed
Note: Files which are to be processed using the direct access subroutines will always produce this message. Since the clause is not then necessary, the error message can be ignored.		
929 RANDOM BUT NO ORGANIZATION	There is no ORGANIZATION clause for a file with ACCESS RANDOM.	ORGANIZATION INDEXED is assumed.
930 NO APPLY	TRANSFER-REPLY or INDEX-BUFFER has been met other than in an APPLY clause.	The remainder of the sentence is ignored.
931 APPLY ALONE	APPLY is not followed by TRANSFER-REPLY or INDEX-BUFFER.	The remainder of the sentence is ignored.
932 ACCESS ALONE	ACCESS is not followed by RANDOM or SEQUENTIAL	The remainder of the sentence is ignored.
934 NO ORGANIZATION	DIRECT or INDEXED are not preceded by ORGANIZATION.	The remainder of the sentence is ignored.

Message	Cause	Compiler action
935 LONE INVALID	INVALID is not preceded by READ, WRITE or DELETE.	The remainder of the sentence is ignored.
936 ORGANIZATION ALONE	ORGANIZATION is not followed by DIRECT, INDEXED, FIRST-LEVEL or SECOND-LEVEL. Note: In #XE20 this message still appears if the word FIRST-LEVEL or SECOND-LEVEL appears in this clause	The remainder of the sentence is ignored.
937 PROCESSING ALONE	PROCESSING is not followed by SEQUENTIAL.	The remainder of the sentence is ignored.
938 RANDOM ALONE	RANDOM is not preceded by ACCESS.	The remainder of the. sentence is ignored.
939 SEQUENTIAL ALONE	SEQUENTIAL is not preceded by PROCESSING or ACCESS.	The remainder of the sentence is ignored.
950 FILE UNASSIGNED IN APPLY	A file referenced in an APPLY statement has not appeared in a SELECT....ASSIGN statement.	The file reference is ignored.
951 FILE AAAA ALREADY EXISTS	A new Steering Line has been read for an existing file, requesting relabelling to the current file name.	The new Steering Line is ignored.
952 NO SUBFILE BBBB/*NNN	A subfile, name BBBB and generation number NNN, cannot be found on this file.	The current program is abandoned.
953 SUBFILE LIST OVERFLOW	A compiler list, containing subfile names and generation numbers, has overflowed.	The final list of subfiles on the new *COBOL tape will be incomplete.
954 CORRECT/COPY IN BATCH COMPILE PARAMETERS	After a Batch Compile run has started, a *CORRECT Steering Line has been met.	The correct program is abandoned.
955 TRANSLATE NOT IMPLEMENTED	Translated from RAPIDWRITE has been requested.	The batch run is terminated.

Code list: reference number definitions for error messages 006, 011, 012

Reference number	Type of statement or sentence containing invalid material
0100	(_.....)
0110	ACCEPT_
0123	ACCEPT FILE KEYS FROM_
0127	ACCEPT_FROM_
0135	See 0139, 0144, 0150, 0156 and note on page 237.
0139	ADD 0135 GIVING_
0146	ADD 0135 GIVING_ROUNDED
0150	ADD 0135 GIVING_ROUNDED SIZE ERROR_ 0194
0152	ADD 0135 GIVING_SIZE ERROR_ 0194
0160	See 0165, 0170, 0176, 0182 and note on page 237.
0161	See 0165, 0170, 0176, 0182 and note on page 237.
0165	ADD 0161 [TO] 0160
0170	ADD 0161 [TO] 0160 ROUNDED
0176	ADD 0161 [TO] 0160 ROUNDED SIZE ERROR 0194
0182	ADD 0161 [TO] 0160 SIZE ERROR 0194
0194	See 0150, 0152, 0176, 0182, 0428, 0434, 0445, 0451, 0713, 0719, 0730, 0736, 1210, 1216, 1227, 1233 and note on page 237.
0200	ALL_
0209	_ALPHABETIC
0218	ALTER_TO PROCEED TO_ [_TO PROCEED TO_]
0232	_AND_
0250	BIT-nn [0263] [0274]
0263	ON STATUS IS_ see 0230 and note on page 237.
0274	OFF STATUS IS_ see 0230 and note on page 237.
0292	BLOCK [_TO]_RECORDS
0299	BLOCK [_TO]_CHARACTERS
0307	CARD_PUNCH_[1266]
0314	CARD_READER_[1266]
0320	CHANNEL-n
0329	{CHECK / CHEQUE} PROTECT LEAVING_
0335	CLOSE_
0339	_REEL
0387	DISPLAY_
0390	DISPLAY 0393 UPON_
0393	See 0390 and note on page 237.
0412	DIVIDE_INTO_
0417	DIVIDE_INTO_GIVING_
0422	DIVIDE_INTO_GIVING_ROUNDED
0428	DIVIDE_INTO_GIVING_ROUNDED SIZE ERROR 0194

0434 DIVIDE_INTO_GIVING_SIZE ERROR 0194

0439 DIVIDE_INTO_ROUNDED

0445 DIVIDE_INTO_ROUNDED SIZE ERROR 0194

0451 DIVIDE_INTO_SIZE ERROR 0194

0462 ENTER_ _

0470 See 0472 and note on page 237.

0472 ENTER_USING 0470

0487 – { EQUALS / EQUAL TO / = } –

0491 SUBSCRIPTED NAME

0506 FILE-RECORD

0550 FLOAT SIGN LEAVING_PLACES

0570 See 0576 and note on page 237.

0576 GO 0570 DEPENDING ON_

0583 – { GREATER THAN / EXCEEDS / > } –

0603 See 0606, 0614 and note on 237.

0606 IF (condition) 0603

0612 See 0614 and note on page 237.

0614 IF (condition) 0603 { ELSE / OTHERWISE } 0612

0632 _IS_

0659 – { LESS THAN / < } –

0663 _LOCK

0674 MEMORY (SIZE)_

0677 MEMORY (SIZE)_WORDS

0686 See 0688 and note on page 237.

0688 MOVE_TO 0686

0697 MULTIPLY_BY_

0702 MULTIPLY_BY_GIVING

0707 MULTIPLY_BY_GIVING_ROUNDED

0713 MULTIPLY_BY_GIVING ROUNDED SIZE ERROR 0194

0719 MULTIPLY_BY_GIVING_SIZE ERROR 0194

0724 MULTIPLY_BY_ROUNDED

0730 MULTIPLY_BY_ROUNDED SIZE ERROR 0194

0736 MULTIPLY_BY_SIZE ERROR 0194

0740 _NEGATIVE

0754 _NO REWIND

0764 NOT_

0771 _NUMERIC

0778 OBJECT-COMPUTER._

0784 OCCURS_

0787 OCCURS_TIMES

0801 OPEN [0806] [0811]

0806 INPUT_....
0811 OUTPUT_....
} See 0801 and note on page 237.

0821 _OR_

0827 PAPER_PUNCH_[1266]

0834 PAPER_READER_[1266]

0849 PERFORM_[THRU_]

0861 PERFORM_[THRU_]_TIMES

0869 PERFORM_ THRU_ UNTIL_

0889 { VARYING / AFTER } _FROM_BY_UNTIL_ See 0892 and note on page 237.

0892 0889 [0889 ....] See 0896 and note on page 237.

0896 PERFORM_ [THRU_] VARYING 0892

0902 PICTURE_

0910 POINT LEFT_

0915 POINT RIGHT_

0918 POINT LEFT_BITS

0923 POINT RIGHT_BITS

0925 _POSITIVE

0930 PRINTER_[1266]

0970 RANGE_

0990 See 0992 and note on page 237.

0992 READ_INTO AT END 0990

1004 See 1006 and note on page 237.

1006 READ_AT END 1004

1014 RECORD [_TO] _[CHARACTERS]

1024 RECORDING MODE_

1039 REDEFINES_

1053 RENAMES_

1059 RENAMES_THRU_

1073 RERUN ON TAPE-*_REEL_

1077 RERUN ON TAPE-*_RECORDS_

1081 RERUN ON_REEL_

1085 RERUN ON_ _RECORDS_

1089 RERUN REEL_

1093 RERUN_RECORDS

1098 RESERVE_ [ALTERNATE AREA] See 1140 and note on page 237.

1109 SAME AREA_

1113 SAME RECORD AREA_

1122 _SECTION[_]

1131 SEGMENT_LIMIT_

1140 SELECT_ASSIGN_[1098]

1155 SIZE_[CHARACTERS]

1158 SIZE_

1165 SOURCE-COMPUTER._

1179 STOP_

1190 See 1194, 1199, 1204, 1210, 1216, 1221, 1227, 1233 and note below.

1194 SUBTRACT 1190 FROM_

1199 SUBTRACT 1190 FROM_GIVING_

1204 SUBTRACT 1190 FROM_GIVING_ROUNDED

1210 SUBTRACT 1190 FROM_GIVING_ROUNDED SIZE ERROR 0194

1216 SUBTRACT 1190 FROM_GIVING_SIZE ERROR 0194

1221 SUBTRACT 1190 FROM_ROUNDED

1227 SUBTRACT 1190 FROM_ROUNDED SIZE ERROR 0194

1233 SUBTRACT 1190 FROM_SIZE ERROR 0194

1256 _THRU_

1266 TYPE_

1276 _UNEQUAL TO_

1296 VALUE_.....

1306 VALUE OF { ID / IDENTIFICATION } _

1309 VALUE OF { ID / IDENTIFICATION } _ACTIVE-TIME_

1322 WRITE_

1327 WRITE_FROM_

1331 WRITE_FROM_BEFORE_

1335 WRITE_FROM_AFTER

1339 WRITE_BEFORE_

1343 WRITE_AFTER_

1360 ZERO SUPPRESS LEAVING_

Note: A number underlined, e.g. '0806' indicates a variable-length list of operands (either names or phrases) and the number refers to the reference number which will appear in the error message when this list is incomplete or invalid. A line without a number, e.g. '____' represents one operand.

For example if the programmer wrote

OPEN NEW-FILE

compiler would print message 012 INVALID SYNTAX 0801.

On referring to the Code List on page 236 against entry 0801 one finds 'OPEN[0806] [0811]', and against the entries 0806 and 0811

'INPUT_*....'

'OUTPUT_*....'

The error message is thus signalling the fact that either the word INPUT or the word OUTPUT has been omitted from the OPEN phrase.

XE20 Code list for error messages 006, 011, 012

The following list applies only to XE20. The list beginning on page 234 of this manual should continue to be used for other COBOL compilers, and does *not* apply to XE20.

Note : The note referred to in the list below is the note on page 237.

Reference number	Type of statement or sentence containing invalid material
0021	Procedure Division sentence
0030	Data Division sentence
0052	Data Division sentence
0055	Data Division sentence
0070	Environment Division sentence
0078	procedure-name declaration
0103	(-.....) - subscripted condition-name
0106	See 0109 and note
0109	(_.....) - subscripted name
0112	- { EQUALS / EQUAL TO / = } -
0117	ACCEPT_
0130	ACCEPT FILE KEYS FROM_
0134	ACCEPT_FROM
0146	ACTUAL [KEY] [IS]_
0153	See 0157, 0162, 0150, 0152 and note
0157	ADD 0153 GIVING_
0162	ADD 0153 GIVING_ROUNDED
0168	ADD 0153 GIVING_ROUNDED SIZE ERROR_ 0212
0174	ADD 0153 GIVING_SIZE ERROR 0212
0177	See 0183, 0188, 0194, 0200 and note
0180	See 0183, 0188, 0194, 0200 and note
0183	ADD 0177 (or 2789) TO 0180 (or 2788)
0188	ADD 0177 (or 2789) TO 0180 (or 2788) ROUNDED
0194	ADD 0177 (or 2789) TO 0180 (or 2788) ROUNDED SIZE ERROR 0212
0200	ADD 0177 (or 2789) TO 0180 (or 2788) SIZE ERROR 0212
0212	0168, 0174, 0194, 0200, 0523, 0529, 0540, 0546, 0853, 0859, 0870, 0876, 1385, 1391, 1402, 1408, and note
0218	ALL_
0227	_ALPHABETIC
0236	ALTER_TO PROCEED TO_[TO PROCEED TO.]
0250	_AND_

0267 See 0276 and note

0274 See 0276 and note

0276 APPLY 0267 INDEX-BUFFER 0274

0284 APPLY FILE-MESSAGES ON . . .

0287 See 0296 and note

0294 See 0296 and note

0296 APPLY 0287 TRANSFER-REPLY 0294

0317 ASSIGN OBJECT-PROGRAM _

0326 BIT-nn [0339] _

0329 BIT-nn [0339] [0350]

0339 ON STATUS IS _

0350 OFF STATUS IS _

0368 BLOCK _[TO]_RECORDS

0370 BLOCK _[TO]_RECORDS

0375 BLOCK _ TO _ CHARACTERS

0377 BLOCK _ TO _ CHARACTERS

0393 CARD-PUNCH _ 1448

0396 CARD-PUNCH _ 1448

0400 CARD-READER _ 1448

0403 CARD-READER _ 1448

0406 CHANNEL-n IS _

0415 { CHECK / CHEQUE } PROTECT LEAVING

0421 CLOSE _

0424 _ REEL

0465 DELETE _

0471 DELETE _ INVALID (statements)

0482 DISPLAY _

0485 See 0488 and note

0488 DISPLAY 0485 UPON _

0507 DIVIDE INTO _

0512 DIVIDE INTO GIVING _

0517 DIVIDE INTO GIVING ROUNDED

0523 DIVIDE INTO GIVING ROUNDED SIZE ERROR 0212

0529 DIVIDE INTO GIVING SIZE ERROR 0212

0534 DIVIDE INTO ROUNDED

0540 DIVIDE INTO ROUNDED SIZE ERROR 0212

0546 DIVIDE INTO SIZE ERROR 0212

0551 EDS _

0563 ENTER _ _

0571 See 0573 and note

0573 ENTER_[USING] 0571

0588 {EQUALS / EQUAL TO}

0605 EXAMINE_TALLYING_

0612 EXAMINE_TALLYING_REPLACING

0628 EXAMINE_REPLACING_BY_

0638 FDS_

0646 FILE-RECORD

0664 FLOAT SIGN LEAVING_PLACES

0684 See 0690 and note

0690 GO 0684 DEPENDING ON_

0693 GO TO

0697 {GREATER THAN / EXCEEDS / >}

0733 INPUT (list)

0739 {I-O / INPUT-OUTPUT} (list)

0755 _IS_

0783 -{LESS THAN}-

0792 _LOCK

0795 CLOSE ---- REEL LOCK

0801 MCF_

0809 MEMORY (SIZE)__

0812 MEMORY (SIZE)_WORDS

0821 See 0823 and note

0823 MOVE_TO 0821

0837 MULTIPLY_BY_

0842 MULTIPLY_BY_GIVING

0847 MULTIPLY_BY_GIVING_ROUNDED

0853 MULTIPLY_BY_GIVING ROUNDED SIZE ERROR 0212

0859 MULTIPLY_BY_GIVING_SIZE ERROR 0212

0864 MULTIPLY_BY_ROUNDED

0870 MULTIPLY_BY_ROUNDED SIZE ERROR 0212

0876 MULTIPLY_BY_SIZE ERROR 0212

0880 _NEGATIVE

0892 _NO REWIND

0902 NOT_

0909 _NUMERIC

0916 OBJECT-COMPUTER._

0922 OCCURS_

1494	VALUE OF { ID / IDENTIFICATION }-ACTIVE-TIME
1497	(FD description)
1499	(FD description)
1502	(FD description)
1504	(FD description)
1515	WRITE_
1520	WRITE_FROM_
1526	WRITE_INVALID (statement)
1532	WRITE_FROM_INVALID (statement)
1536	WRITE_FROM_BEFORE_
1540	WRITE_FROM_AFTER
1544	WRITE_BEFORE
1548	WRITE_AFTER_
1565	ZERO SUPPRESS LEAVING_
2274	IF (Condition) 2287
2279	IF (Condition) 2287 { ELSE / OTHERWISE } 2287
2287	See 2274, 2279
2788	See 0183, 0188, 0194, 0200
2789	See 0183, 0188, 0194, 0200

OTHER COMPILE TIME MESSAGES

A number of other messages may be displayed on the console during compilation. These are chiefly concerned with loading, unloading and allocating peripherals, e.g. when tape changes are required because no additional tape stations have been made available to the compiler for such files as Library Subroutines, user subroutines etc.

A list of these messages follows.

Console message	Cause	Operator action
DISPLAY – 624 UNLOAD UNIT xx	Tape change or other peripheral unload required during compilation.	As printed (xx is absolute unit number).
DISPLAY – 625 UNLOAD UNIT xx LABELxxxxxxxxxxxx	Peripheral unload required during compilation (paper tape or cards).	Unload peripheral and label file as printed (xx is absolute unit number).
HALTED – 626 PRINTER FAILURE	A transfer failure has occurred, on three repeats.	Engineer action required.
DISPLAY – 627 PAPER LOW	As printed.	Reload printer.
HALTED – 629 UNLOAD UNIT xx: LABEL #xxxx (CONT): GO	Consolidated object program to be unloaded from specified peripheral (precedes 643).	As requested. (Note that 'CONT' in the message means this part of output, though produced first, follows the output signalled by 643, on loading at run time – see page 247.) xx – absolute unit number.
HALTED – 630 UNITXX: CHECK-SUM ERROR RE FEED/ ABANDON	Checksum errors on reading user subroutines from cards, paper tape or magnetic tape during input to Consolidate Phase.	Refeed offending card or block up to three times, then abandon if same message is printed.
HALTED – 643 UNLOAD UNIT xx LABEL #nnnn	Consolidated semi-compiled object program to be unloaded and labelled (cards or paper tape).	As printed.
DELETED – LOAD #nnnn on xx	Consolidated semi-compiled object program to be loaded on specified peripheral.	As printed (xx is absolute unit number).
DISPLAY – 651 PUT xxxxxxxxxxxx ON UNIT nn	Slow peripheral input unit is to be loaded, e.g. source program, Steering Lines, on card reader, paper tape reader.	Load the unit as requested. (x .. x = file name).

Note: The following group of messages, 652 to 660 and 662, may result from incorrect peripheral requests by the programmer in his Job Sheet. These errors can be corrected by the given operator action during compilation, but for the next compilation, the programmer should examine whether, in the light of the message, he should change his peripheral requests.

Console message	Cause	Operator action
HALTED - 652 ALLOCATE CR: PUT ON xxxxxxxxxxx: GO	Card reader required for compilation.	Allot reader, place named file in reader and GO.
HALTED - 653 ALLOCATE TR: PUT ON xxxxxxxxxxx: GO	Paper tape reader required for compilation.	Allot reader, place named file in reader and GO.
DISPLAY - 654 UNLOAD UNIT ---	Specified unit to be unloaded.	As requested.
DISPLAY - 655 MOUNT MT* xxxxxxxx WITH WPR	Magnetic tape unit with write permit ring to be loaded with tape serial number xxxxxxxx (O/P tape).	As requested. If there is no spare deck and the Master Library Tape is not allocated to the compiler, it should be removed, and that deck used.
DISPLAY - 656 MOUNT MT* xxxxxxxx NO WPR	As above, but without write permit ring (I/P tape).	As requested. If there is no spare deck and the Master Library Tape is not allocated to the compiler, it should be removed, and that deck used.
DISPLAY - 657 MOUNT SCRATCH TAPE WITH WPR	Scratch tape required for compilation.	As requested.
HALTED - 658 ALLOCATE CP: GO	Card punch required for compilation.	As requested.
HALTED - 659 ALLOCATE LP: GO	Line printer required for compilation.	As requested.
HALTED - 660 ALLOCATE TP: GO	Paper tape punch required for compilation.	As requested.
HALTED - 662 filename (generation number) EDS ERR-SEE LP	The filename quoted is that of the file in error.	Reference to the printer will show the error type.
DISPLAY: NEEDS AAAA	Subroutine or cue name AAAA has not been found in consolidation.	Program is consolidated leaving AAAA as a blank cue.
DISPLAY: TAPE ERROR (UNIT NN)	An irrevocable tape failure has occurred.	The current program is abandoned.
HALTED: LOAD DOCUMENT AAAA	A document line on the front of a slow peripheral file has been read with a name not equal to that given in the appropriate Steering Line.	Load correct file and GO.
HALTED: NOT STEERING LINES FILE	As above, a document header other than DOCUMENT. STEERING has been read on the slow peripheral Steering Lines file, or the first line does not begin with an asterisk.	Load correct Steering Lines file and GO.
DISPLAY: PROGRAM AAAA ABANDONED	Due to a restart (GO 26) the current program has been abandoned.	Compiler goes on to next program.

TERMINAL MESSAGES OF COMPILATION

Printer

STATISTICS: *DATA nnnnn X pp.g
STATISTICS: *PROCEDURE rrrrr X ss.t
STATISTICS: *LOWER = nnnn (this line is printed by the Consolidate Phase).

where nnnnn is the number of data names in the source program.

pp.g is the average number of characters in the data names which does not exceed 30.0.

rrrrr is the number of procedure names in the source program.

ss.t is the average number of characters in the procedure names in the source program, which does not exceed 30.0.

nnnn is the amount of Lower required by the object program less the amount of Lower required by Data Division fields, as shown in the Data Map listing.

If the store limits have been exceeded, one or both of the following messages may be printed:

ERROR REF 200 LOWER STORAGE EXCEEDED BY nnnnn WORDS

ERROR REF 210 OBJECT PROGRAM HAS EXCEEDED COMPUTER SIZE BY nnnnn WORDS

After these messages the Consolidator Listings are printed if the *CONSOLIDATE Steering Line has been used, for which see page 190.

Note that some of these messages may change or be omitted in later versions of the compilers.

Console typewriter

Depending on the mode of compilation, one of the following messages will be typed:

HALTED-SERVICED ppppnnnnnnnn

or HALTED-COMPILED ppppnnnnnnnn

or HALTED-CONSOLIDATED ppppnnnnnnnn

where pppp is the COBOL program name and nnnnnnnn is the Account Number.

If any ERRORS (as opposed to CAUTIONS) have been flagged during the run, the additional legend "(nnnn ERRORS)" will appear after the standard terminal message, where nnnn is the number of such errors found.

Chapter 17 The object program

FORMAT

If the *CONSOLIDATE Steering line has not been used, the object program will be in standard 1900 semi-compiled form, suitable for consolidation later, but not for loading and performing.

If *CONSOLIDATE has been used, the object program will be in standard consolidated form. From the user's point of view, therefore, there is no difference between running an object program whether produced by a PLAN or COBOL compilation.

If the COBOL source is divided into overlay segments, the loader will automatically give a binary dump to magnetic tape. If the COBOL source is not an overlay program, and a binary dump is required, it must be done by standard operator action at run time, via Executive.

The format of a consolidated object program on magnetic or paper tape or cards is:

Program blocks

Consolidated leaders

Tape mark (magnetic tape only)

Load request slip (approximately 15 words)

Loader (overlay or non-overlay)

Entry block

ALLOCATION OF STORAGE IN THE OBJECT PROGRAM

The following is an example of the way core store is used by a COBOL object program compiled by the XE13 Compiler. The source program was unsegmented and had three magnetic tape files without "Alternate Areas". The Terminal Messages and Consolidator listings were:

```
MAP SECTION COBOL                                    PERMANENT PROGRAM   1799
STATISTICS:   *DATA        =    560 x 11.2
STATISTICS:   *PROCEDURE   =     53 x 11.3
STATISTICS:   *LOWER       =    518 WORDS
CORE          7872
                LV 45              LP 246     LT 1935      UP 5365      UV 5598
MAP PERMANENT PROGRAM BEGINS AT 2043
41              5264    COBL
41              4653    HMTWR
41              4695    HMTRD
41              5235    COBO
41              5254    COBI

41              4634    HMTEND
41              5014    HMTDEF
41              5221    HMTMOD
```

41	4627	DATEPLEASE	User's Own Code routines.
41	3842	COBOLPRINT	
41	3855	CLOSEPRINT	
41	3902	PRINTDETAIL	
41	4766	HMTP	
41	4956	HMTF	
41	4754	HMTN	
41	4926	HMTD	
41	4805	HMTQ	
41	4789	HMTO	
41	4712	HMTM	
41	4819	HMTC	
41	4844	HMTE	

From these messages the complete layout of the object program in store may be constructed:

Words	Usage
0-44	Reserved (e.g. 0-7 are Accumulators, 20-29 are standard Entry Points, etc.).
(*Lower variable*	always begins at Word 45.)
45-76	Always allocated in object programs as a "Scratch Pad" area used e.g. as a print line for DISPLAYed messages, for subscript calculations etc. (Note that later compilers will generate larger Scratch Pad areas.)
77-167	Record areas, each 30 words long, for the 3 magnetic tape files defined in the source program
168-245	Lower Variable requirements of MTH routines, Own Code routines etc.
(*Lower preset*	begins, for this program, at Word 246.)
246-297	Standard run-time constants and the Run-Time File Table (3 words per file).
298-1541	WORKING-STORAGE areas.
1542-1934	Compiler-generated constants and LP requirements of subroutines (MTH Own Code etc.)
(*Literals*	begin for this program, at Word 1935.)
1935-2042	Literals as required by run-time Subroutines COBI, COBL etc, and Own Code routines.
2043-3841	Object code for COBOL PROCEDURE DIVISION.
3842-4633	User's Own Code routines.
4634-5234	MTH routines
5235-5364	Run-time routines COBO, COBI, COBL.
(*Upper preset*	begins, for this program, at Word 5365.)
5365-5597	File Definition Area entries (30 words for each magnetic tape file): certain compiler-generated literals, and UP requirements of run-time S/Rs etc.
(*Upper variable*	begins, for this program, at Word 5598.)
5598-7872	File area (sometimes called Block area or Buffer area) for each magnetic tape file, 750 words per file (as each block contained 25 records) = 2250 words. Of the remaining 24 words 6 are reserved, and 18 round up the total core store requirement of the object program to a multiple of 64.

The *LOWER steering line

This option enables the user to restrict, as far as possible, the amount of Lower Storage allocated in

the object program *by the compiler* It is used principally after an initial compilation when messages 200 LOWER STORAGE EXCEEDED [BY nnnn WORDS] or 114 (or 714) NOT ENOUGH LOWER or message 620 O/P LOWER nnnnnn WORDS TOO LARGE have been printed.

The allocation of Lower in the object program is of three types:

1 Storage allocated to Data Division fields (during compilation).

2 Storage allocated to user's program literals (literals quoted in the Procedure Division) and to compiler-generated literals required for procedural statements (during compilation).

3 Lower Storage requirements of user's Own Code (ENTERed) routines, MTH routines and compiler-generated run-time routines such as COBA, COBB etc. (allocated during Consolidation).

Savings in types 2 and 3 can only be achieved by the user in the way his COBOL program is written (see page 216, message 210).

Savings in type 1 *can* be achieved by the compiler provided it is told in a * LOWER Steering Line how much Lower storage must be reserved for allocations of types 2 and 3. This figure will be derived from messages printed during the previous compilation when the compiler signalled that Lower storage had been exceeded. This situation can arise at different points in compilation, as described below.

1 If message 620 was printed during Consolidation, i.e. after the CORE line, then the message STATISTICS: * LOWER nnnn WORDS will also have been printed, immediately before the CORE line. The figure in the STATISTICS line should then be quoted in the *LOWER Steering Line for the next compilation, being the total Lower requirements of types 2 and 3.

2 Message 114 or 714 may be printed during the Analysis Phase, when storage is being allocated to fields in the Data Division (type 1). In this situation the compiler may stop before the Generate Phase, the Terminal Messages may not be printed and if so the user will not know exactly how much Lower allocation types 2 and 3 will require. He should therefore quote the maximum permitted figure, 3500, in his *LOWER Steering Line for the next compilation, which will have the effect of forcing all or most of his data fields into Upper. This compilation will also give him a fresh set of allocation figures, so enabling him, if he wishes, to reduce the *LOWER figure for later compilations until he achieves total utilization of Lower. If Lower is still exceeded after quoting a figure of 3500 in the Steering Line, it means that the source program cannot be compiled and consolidated successfully as written.

   Message 114 or 714 may be printed as an ERROR or a CAUTION. In the former case it means that Lower has already been exceeded; in the latter that seven-eighths of available Lower has been allocated (and thus there will almost certainly not be enough remaining for allocation types 2 and 3).

3 Message 200 may be printed during the Generation Phase. The translation of the COBOL source into object code continues, but it is not *generated* (i.e. output). At the end of Generate Phase message 200 will be printed again, this time with the addition of "BY nnnn WORDS". nnnn added to 4096 gives the total of allocation types 1 and 2, say, N. From the Data Map listing the figure for type 1 allocation may be calculated, say, M. Then N - M is the figure for type 2. The figure for type 3 will not be known, since compilation will have terminated before the Consolidate Phase is entered, and the STATISTICS: *LOWER line will not have been printed. The user can probably make a reasonable estimate, bearing in mind that for most programs MTH and run-time routines will need about 100 to 250 words of LOWER. To this must be added the Lower requirements of the user's Own Code routines.

   If the estimate is P, then the figure to be quoted in the *LOWER Steering Line for the next compilation is N - M + P. However, the user can avoid complications in this situation by always quoting the maximum permitted figure, 3500.

Note: Reference was made above to the "maximum permitted figure, 3500" in a *LOWER Steering Line. This figure is not invariable but may be taken as valid for most occasions.

PERIPHERAL HANDLING IN THE OBJECT PROGRAM

When the COBOL input/output verbs, OPEN, READ, WRITE, CLOSE refer to magnetic tape files, the compiler generates calls to routines in the 1900 Magnetic Tape Housekeeping (MTH) package. The required routines are incorporated in the object program once only, in the permanent part of the program if the COBOL source is divided into overlay segments (see also note (i) on the ENTER verb on page 144).

All other peripheral handling is accomplished by instructions, control areas and characters generated by the compiler.

The object program will always contain a File table of three-word entries for each file described in the Data Division. The contents of these entries will vary at run-time according to whether a file is input or output, opened or closed, and whether end-of-file has been reached; and the object code tests whether the file is in a valid state for the particular input/output instruction about to be performed. The entries also contain dump information if there was a RERUN statement in the COBOL source. The File table is in Lower Preset.

If any files in the COBOL source have been assigned to magnetic tape, the object program will contain a File Definition Area with 30-word entries for each file so assigned. The area is in Upper Preset and in the format required by the Magnetic Tape Housekeeping package.

If there was a RERUN statement in the COBOL source, the object program will contain a Dump Parameters table of length n + 3 words, where n is the number of files described by the user in his Data Division. This table is in Lower Preset and is used by the run-time Dump package.

Logic of the instructions generated for peripheral handling

The following analysis of the code generated for COBOL input/output statements by one version of the compiler is given for those users who may be interested in the complete system of data file handling in 1900 COBOL object programs.

abbreviations: RTFT = Run-Time File Table
DPT = Dump Parameters Table
FDA = File Definition Area
MTH = Magnetic Tape Housekeeping package

Names beginning COB refer to run-time routines incorporated in the object program. Underlined text refers to the action of the object code, the remainder to compiler action.

COBOL verb		Compiler action and generated object code
OPEN (character peripherals)	1	CALL COBL.
	2	CALL COBS.
	3	If double-buffered input file read one record.
	4	If dumping is required, move dump parameters from RTFT to DPT.
OPEN (magnetic tape)	1	CALL COBL.
	2	Set word 5 of FDA = 0.
	3	If dumping is controlled by this file at EOR then: for O/P files set word 10 of FDA = 0 for I/P files set word 10 of FDA = Word 0 of LV
	4	If generation no. check is required extract generation-no. from RTFT; if it is zero CALL COBM and store generation no. in FDA.
	5	CALL MTMOD (MTH Macro to set MODE in FDA).
	6	CALL MTDEF (MTH Macro to open a file) Among the parameters is one to set word 11 of FDA to address of COBN if this is an O/P file controlling dumping; otherwise word 11 = 0
NOTE No check is made for the presence or absence of a WPR.	7	Set character 13.0 of FDA = 2 for O/P files. or = 3 for I/P files.
	8	If it is an output file, zeroize first word of record.
	9	If dumping is required, move dump parameters from word 1 of RTFT to DPT.
READ (character peripherals)	1	CALL COBL.
	2	Read 1 Record.
	3	If this file controls dumping 'every n records' count records and do a dump via COBN if necessary.

COBOL verb	Compiler action and generated object code
	4 If end-of-file has not been reached, exit, otherwise, 5 set end-of-file indicator in RTFT. 6 If AT END statement is a GO, generate a branch to the stated paragraph preceded by a branch round this instruction, otherwise, generate instruction to branch to a compiler-created label that follows coding for all the statements in the AT END branch.
READ (magnetic tape)	1 CALL COBS. 2 Store address of step 7, in object coding in word 11 of FDA. This is the address to which the MTH Read routine exits if there is an 'exception'condition' during reading (see note below). MTH leaves in the object program's word 11 a parameter giving the reason for exit, and a return address to itself. 3 CALL MTRD (MTH macro for reading magnetic tape). 4 If this file controls dumping 'every n records' count records and do a dump via COBN if necessary. 5 Branch round AT END procedures. (Step 4 or 5 is the normal return address from the macro if no exception condition has arisen.) 6 Take appropriate action according to exception condition. Only if the exception condition is 'end-of-file' does the object program fall through to 8. 7 and 8 – as 5 and 6 of READ character peripherals. Note: The various types of 'exception conditions' and the action taken by the object program are: LONG BLOCK A block has been read which exceeds the buffer size. The program will be suspended and no restart is possible. END OF FILE End procedure is intiated, per user's instructions in AT END clauses. USER SENTINEL User Sentinels cannot at present be written in 1900 COBOL but if the tape was created by, say, a PLAN program, there may be User Sentinels on the tape. They will be ignored and the program will return to MTH to read another block. UNCLEARABLE PARITY ERRORS Executive has been unable to clear a parity failure and suspends the program with a FAIL message. The operator has restarted the program and MTH takes no action other than exiting. END OF REEL 1900 COBOL programs only use this exit if the file being read controls dumping, at EOR, in which case dumping is initiated whenever the end of a reel of the file is reached. The program returns to MTH to read another block. The code strings generated for examining exception conditions differ according to whether or not the file controls dumping at EOR. If it does, and end-of-reel has been found, a CALL is generated to COBN.
WRITE (character peripherals)	1 CALL COBL. 2 If it is a print file, CALL COBQ, or COBR for 'advancing'. 3 Write one record.

COBOL verb		Compiler action and generated object code
WRITE (magnetic tape)	1	CALL COBL.
	2	If first word of record is zero, store in it the size of the record in words.
	*3	If dumping is required and this is the file controlling dumping, at EOR, set parameters in X6, X7 (required by the run-time dump routine COBN as a parameter for the dump package).
	4	CALL MTWR (MTH macro to write a magnetic tape record).
	5	Zeroize first word of record.
CLOSE (character peripherals)	1	CALL COBL.
	2	If the file is assigned to card or paper tape punch, punch two end-of-file markers (****) in columns 1 to 4.
	3	If 'LOCK' is specified DISENGAGE peripheral.
	4	Release peripheral from program.
	5	If dumping is required, zeroize the file entry in DPT.
CLOSE (magnetic tape)	1	CALL COBL.
	2	CALL MTEND (MTH macro to close the file).
	3	As *3 above, and in addition, if this is the file controlling dumping at EOR CALL dump routine COBN.
CLOSE .... REEL (magnetic tape)	1	CALL COBL.
	2	If this file controls dumping at EOR, prime COBN.
	3	CALL MTCRE (MTH Macro to close a reel).
	4	If this file controls dumping at EOR, and if it is an I/P file, CALL COBN.

Labels and sentinels

All header labels written or checked by compiler-generated routines (called from the Housekeeping packages) will/must be in standard 1900 format. User-defined labels are not permitted but the user does have the option of checking the generation number by specifying LABEL RECORDS ... WITH GENERATION-NO in the File Description (see page 37).

The compiler-generated routines will process all standard sentinels in accordance with the rules of the Housekeeping packages but will ignore user sentinels.

The processing of Trailer labels is not permitted in 1900 COBOL but the facility in the Housekeeping packages is nevertheless used as a means of finding the end of a reel of a multi-reel file, when such a file controls dumping.

Dumping and restarts

Dumping is achieved in COBOL by a RERUN statement in the Environment Division. There can be only one such statement in a program. If the dump is to a user-defined file, it must be a magnetic tape and open for output whenever a dump is to be taken. A tape created in this way will have the user's data and dump information on it. When this tape is read as a data tape the Magnetic Tape Housekeeping package will ignore the dump information and the user need not be aware that this information is on the tape. A Restart from this tape is achieved in the same way as if the dumps were on a special tape. The RERUN statement is all that is required to be written by the user to achieve dumping.

At the appropriate places in the code generated for COBOL READ/WRITE verbs referring to the file controlling dumping, CALLS are generated to the subroutine COBN. This subroutine in turn calls the Dump package which the consolidator incorporates from the master library tape or other input medium in the object program, as described in chapter 17.

When a dump has been completed at object-run-time the program halts and the message 'ATTENTION TO STATE OF SLOW PERIPHERALS' is displayed on the console. The operator should mark up the dump point on the slow peripheral data and restart by typing GO.

For the procedure to reload an object program from a dump using the restart routine see the Library Specifications and the Magnetic Tape Manual. If the RERUN statement quoted EDS-* or FDS-*, a file with the name ICT-DUMPFILE (*not* ICL-DUMPFILE) should be allocated before the object run. Information about dumping in direct access environments may be found in the 1900 series manual *Direct Access.*

For further information on Dumps and Restarts see the subroutine COBN on pages 252 to 253.

File numbering in the object program

Three sets of numbers are involved, and should not be confused:

1 For files assigned to basic peripherals and disc units the user writes in the COBOL Environment Division, e.g.

SELECT CARD-INPUT ASSIGN CARD-READER nn

The compiler takes nn as the program unit number of the card reader to which this file is assigned, and it will be quoted in such run-time Executive messages as

UNIT pp USED AS nn

where pp is the *absolute* unit number of the peripheral in the installation.

2 In the case of magnetic tape files the SELECT ... ASSIGN ... statement does not allow the user a program unit number, and the Compiler therefore assigns numbers, from 1 up, to all magnetic tape files in the order in which they are SELECTed in the Environment Division. These numbers are then quoted in run-time Executive messages, exactly as for slow peripherals.

Note: The number '0' is reserved; when the object program is segmented into overlays it is to 'tape number 0' that it will be copied, in binary, by the Loader (and this tape should be preserved for later object runs). Furthermore, if there has been a RERUN statement in the COBOL source program and the dump is to a magnetic tape, this tape is assigned by the compiler a number one greater than the last COBOL magnetic tape file. Both these points should be borne in mind when assigning numbers to magnetic tape files which are exclusively defined and referenced in users' ENTERed routines.

3 For the purposes of run-time messages such as LOAD FILE nn ON etc., *all* files in the COBOL source program are also numbered, *from 0 up,* in the order in which they are declared in the Data Division.

OPERATING INSTRUCTIONS FOR THE OBJECT PROGRAM

Running a consolidated object program

1 When the object program is on cards or paper tape, it is loaded by typing LO #cccc and entered by typing GO #cccc 20 (where cccc is the program name).

2 When it is on magnetic tape, it is loaded by typing FI #cccc and entered as in 1.

There is thus little difference between running a consolidated object program whether it is the end-product of a COBOL compilation or a PLAN compilation, except where the compiler-generated halts are concerned. Full details of the run-time subroutines that incorporate these halts are given on pages 251 to 254. The halts that may occur and the required operator action are as follows:

Run-time messages and halts

Console message	Cause and meaning	Operator action
LOAD FILE nn ON pp	A data file assigned to a slow peripheral is to be opened. (nn – file number pp – absolute unit number)	The file is loaded and the program restarted by typing a GO message.
SET FILE nn GENERATION-NO. IN WORD 0	The generation number of a file to be opened is required. (nn-file number, from Job Sheet)	The operator ALTERs the contents of Word 0, storing in it the number given in the Job Sheet, and restarts by typing GO.

Console message	Cause and meaning	Operator action
CY (or TF	Unclearable parity error on a magnetic tape file.	No restart: abandon run.
LONG BLOCK DETECTED ON FILE nn	A block which is larger than the area specified in the BLOCK CONTAINS clause in the Data Division has been detected.	No restart: abandon run.
COBOL ACCEPT (SEE JOB SHEET) nnnnnn M	Data is to be input to the program from the console. (nnnnnn – sequence number of COBOL sentence M – the number of the ACCEPT statement within the COBOL sentence (see page 72.).)	The operator ALTERs the contents of Word 0, storing in it the value given on the Job Sheet; there may be a number of such values: nnnnnn M gives the one required.
FILE nn NOT CLOSED AT EOR	As printed. The message is printed for each file that is not closed when the STOP RUN statement is obeyed. For output files this means that the end markers will not be present.	None since the program is not suspended.
END OF RUN	Self-explanatory (results from COBOL 'STOP RUN' statement).	No restart.
CZ	An attempt has been made to GO TO an undeclared paragraph, or the program has "fallen through" the last statement in the COBOL source. The halt will not, however, be reached, if any "undeclared paragraph" has been ALTERed.	No restart.
ATTENTION TO STATE OF SLOW PERIPHERALS	Generated by a COBOL RERUN statement. See run-time routine COBN. The message is displayed when a Dump or Restart has been completed.	When a Dump has been completed, the operator records the position of paper tape, last card read etc., and restarts by typing GO. When Restart has been completed, the operator repositions the paper tape, cards etc., in the same positions they were in when the corresponding Dump was made.
DUMMY ROUTINE ENTERED IN SENT nnnnn	A standard routine was entered in place of a user's routine as a result of compilation errors. (See error message 207 on page 216). (nnnnnn – COBOL sentence number).	There is no halt.
DUMP FILE CANNOT BE OPENED	The source program contained RERUN ON EDS-* (or FDS-*) but the dumpfile (ICT-DUMPFILE) cannot be opened at run-time because either a) it is not on-line or	Attempt to re-open by typing GO; if this fails, return to programmer for corrective action

Console message	Cause and meaning	Operator action
	b) its purge date is not exceeded or c) there is an integrity code failure	
QQ	the object program is attempting to branch to a missing segment	Reconsolidate supplying the missing segment

	Message	Meaning	
SENT nnnnnn FILE nn followed by ALREADY OPEN or LOCKED or NOT OPEN or NOT INPUT or READ AFTER EOF or NOT OUTPUT or CLO/REEL AFTER EOF or OPEN (SAME AREA) or WRITE AFTER EOF or DELETE AFTER EOF or NOT I-O or NOT INPUT OR I-O or REWRITE AFTER EOF or EXCEPTION TYPE aa or RETURN AFTER EOF or NOT AT END or NOT REWOUND or RETAINED	ALREADY OPEN	An attempt has been made to open a file that is already open.	No restart: abandon run.
	LOCKED	An attempt has been made to open a file that has been closed with LOCK.	
	NOT OPEN	An attempt has been made to READ, WRITE or CLOSE a file which is not open.	
	NOT INPUT	An attempt has been made to READ a file which is open for output.	
	READ AFTER EOF	An attempt has been made to READ a file after the end has been reached. (The file is still open).	
	NOT OUTPUT	An attempt has been made to WRITE to a file which is open for input.	
	CLO/REEL AFTER EOF	An attempt has been made to CLOSE REEL after the end has been reached. (The file is still open).	

Console message	Cause and meaning		Operator action
	Message	Meaning	
	OPEN (SAME AREA)	An attempt has been made to OPEN a file when another file which uses the same area (see SAME AREA clause on page 32) is open.	
	WRITE AFTER EOF	An attempt has been made to WRITE a record "before the last one read" when the last READ statement indicated End of file.	
	DELETE AFTER EOF	An attempt has been made to DELETE the last record read when the last READ statement indicated End of file.	
	NOT I-O	A DELETE or REWRITE statement has been given to a file that is not open in INPUT-OUTPUT mode.	
	NOT INPUT OR I-O	A SEEK statement has been given to a file that is open in OUTPUT mode.	
	REWRITE AFTER EOF	An attempt has been made to REWRITE the last record read when the last READ statement indicated End of file.	
	EXCEPTION TYPE aa	Where aa is one of the following values: 02 An INVALID KEY condition has occurred on a WRITE or DELETE statement and there is no INVALID KEY clause. 03 File overflow has occurred and there is no TRANSFER-REPLY field. 04 An integrity code check has failed on OPENing a file and there is no TRANSFER-REPLY field. 05 The purge date has not been passed on a file which is to be OPENed for OUTPUT, and there is no TRANSFER-REPLY field.	

Console message	Cause and meaning		Operator action
	Message	Meaning	
		06 A duplicate key condition has arisen on a WRITE statement and there is no TRANSFER-REPLY field.	
		07 A file that is to be OPENed cannot be found and there is no TRANSFER-REPLY field.	
	RETURN AFTER EOF	An attempt has been made to RETURN a file after the end has been reached. (The file is still open)	No restart: abandon run.
	NOT AT END	An attempt has been made to OPEN INPUT REVERSED when no CLOSE NO REWIND has been performed. or to CLOSE NO REWIND an Input file before end-of-file has been reached	No Restart: abandon run.
	NOT REWOUND	A file has had CLOSE NO REWIND applied, and the next Statement applied to it is not OPEN INPUT REVERSED, or CLOSE, or CLOSE (LOCK) (RETAIN)	
	RETAINED	An attempt has been made to SORT a file that has been OPENED, but not CLOSED (without options such as Lock etc.)	
ATTEMPT TO OBEY AN UNALTERED 'GO' SENT nnnnnn	In a GO TO ▫ statement (i.e. no destination paragraph) the code produced by the compiler is a CALL of COBP. The statement should be ALTERed before it is obeyed, but if it is not, COBP is entered and this message is typed. (See also page 88). (nnnnnn – sentence number of – GO TO statement).		No restart: abandon run.
LOAD ACCEPT DATA ON aa	Generated by a COBOL ACCEPT statement when the data is to be ACCEPTed from a paper-tape or card reader. (aa – the absolute unit number).		The operator loads the data as specified in the Job Sheet and restarts by typing GO.
LOAD FILE KEYS ON aa	Generated by COBOL ACCEPT FILE KEYS statement. (aa – absolute unit number).		The operator loads the data as specified in the Job Sheet and restarts by typing GO.

Console message	Cause and meaning	Operator action
ALLOCATE aa TYPE ttt or ALLOCATE aa	A peripheral unit aa (where aa = TR, LP etc.) and of type ttt (see the Properties Code on page 27) is required and is not available. If the TYPE option is not used or TYPE 000 is specified, the second format is printed.	The operator has to make the required peripheral available, or wait for another program to release it and then restart the program by typing GO #name.
GEN-NO. NOT REQUIRED FOR FILE nn	A parameter that has been read by means of an ACCEPT FILE KEYS statement is not required because (a) the WITH GENERATION-NO clause was not specified for the file, or (b) there has already been a generation number for this file from a previous parameter. In case (a) the generation number is ignored; in case (b) the generation number is accepted and replaces the previous one. (nn – file number (see page 247)).	None, the program is not suspended.
INVALID FILE-KEY	A parameter that has been read by means of an ACCEPT FILE KEYS statement is of invalid format or has a file number that exceeds the highest allocated number.	The operator can either repunch the parameter and place it in the card/paper tape reader to be read again, or ignore it completely. To restart the program type GO. If the parameter is ignored, then when the file is OPENed the message SET FILE nn GEN - - - (see page 247) will be typed.

The following error messages are produced by the Magnetic Tape Housekeeping Package, for further details of which see the *Magnetic Tape* manual.

Console message	Cause and meaning	Operator action
MT ERR C	The block size on tape is greater than the area specified by the BLOCK CONTAINS clause in the Data Division. Accumulator 5 contains the unit number assigned to the file.	Output the accumulators 0-7 and abandon the run.
MT ERR F	A record presented for output is longer than the area specified in the BLOCK CONTAINS clause in the Data Division or there is a non-integral number or records in an input block. Accumulator 5 contains the unit number assigned to the file.	Output the accumulators 0-7 and abandon the run.

Console message	Cause and meaning	Operator action
MT ERR G	There is a data block count discrepancy. Accumulator 4 contains the current data block count as accumulated by the Magnetic Tape Housekeeping Package. Accumulator 5 contains the unit number assigned to the file (see page 247). Accumulator 6 contains the data block count from the sentinel on the tape.	Output the accumulators 0-7 and abandon the run.

Exception conditions found when sorting direct access files cause the output of a console message in the format

SORT HALT *xx*

or

SORT ERROR *xx*

where *xx* is a two-character code. SORT HALT indicates that the program may be able to continue after operator action. SORT ERROR indicates that the program cannot continue.

The error codes are listed below in alphabetical order.

Code	Reason	Action
CE	Checksum failure	Abandon run and restart at beginning. If error persists report for investigation.
E1	Number of keys not in range 1-10	Correct and rerun
E2	Key type not in range 1-6	Correct and rerun
E3	Key direction not equal 0 or 1	Correct and rerun
E4	Key length not valid	Correct and rerun
E5	Key address not valid	Correct and rerun
J1	File not in system	Ensure that file is in system and GO to attempt open of file again.
J2	Failure of integrity code check	Suspend program for continuation later.
J3	Insufficient core store for holding further descriptions; or an attempt has been made to open more than the permitted number of files.	Suspend program for continuation later.
J4	Purge date not exceeded	Abandon run
J7	A utility routine was writing to the system control area while trying to open a file.	Suspend program for continuation later.
K0	System control area is full when trying to extend the output file.	Delete any redundant files and GO to attempt extension.
K1	There is no space available to extend output file.	Make space available (for example, load another cartridge, delete redundant files) and GO to reattempt extension.
K3	A non-specific error when trying to extend output file.	If using E.D.S. or twin E.D.S., change cartridge and GO. If error persists or if using F.D.S., report for investigation.
K5	File is not open as a write file when trying to extend output file.	Abandon run. Report error for investigation.

Code	Reason	Action
L0	A non-specific error in trying to read or write a file.	Abandon run. Report error for investigation.
L1	Logical bucket number out of range.	Check work file large enough for input data. If this is not the reason, report for investigation.
L4	Unclearable parity error while trying to read or write a file.	Abandon run.
L8	Auxiliary control area full.	Abandon run.
M5	Bucket sizes of the work files are not equal.	Abandon run
M6	Bucket size of the output file not the same as work files.	Abandon run.
M7	Bucket size of output file not large enough to accept longest record.	Abandon run.
Q1	Failure of internal buffering arrangement of program during pre-string.	Abandon run.
Q2	A record shorter than the minimum length implied by the key definition has been found.	Abandon run.
Q3	As Q2	Abandon run.
Q4	Failure of reload of COBOL program after failed sort.	Abandon run.
R1	Less than 5888 words of core available to sort.	Take appropriate action, for example put correct library on line, delete other programs, etc. and GO. The failed operation will then be repeated.
R2	Insufficient core to reload COBOL program.	
R3	#XSXA not found on direct access Library file.	
R4	#XSXA not found on magnetic tape Library file.	
R5	Open not successful when opening magnetic tape Library.	
R6	File not in system	
R7	Failure of integrity code check	
R8	As J3	
R9	As J7	
S1	Purge date not exceeded	Abandon run. Rerun when error causing fault has been corrected.
S2	Bucket number out of range	
S3	Auxiliary control area full	
S7	No free file numbers	
S8	Record presented to input batcher has zero word count.	
S9	Record presented to input batcher is larger than buffer.	

Code	Reason	Action
ST	Insufficient core for merge	Abandon run. Rerun with larger maximum core or delete other programs and GO.
T0	An attempt has been made to use an output procedure after a failed sort.	Abandon run.
T1	Parity error on magnetic tape or direct access	Abandon run and restart at beginning. If error persists report for investigation.
T2	Checksum failure: reading from magnetic tape Library.	
T3	Checksum failure: reading from direct access Library.	
T4	Checksum failure: COBOL program after reloading.	
T5	Direct Access Library: directory description not found.	Abandon run. Correct library format and rerun.
T6	Direct access: data record not found.	
T7	Magnetic tape Library: data specifying block not found	
T8	Magnetic tape Library: entry block found in wrong place.	
T9	Magnetic tape Library: tape mark found in wrong place.	
W1	At the end of a merge pass the program's control subsets are not reconcilable.	Report for investigation.
W2	At the end of a merge level some records appear to have been lost.	Report for investigation.
W3	During the merge records in a sorted string are found to be out of sequence.	Report for investigation.
W4	Failure in the internal buffering arrangement of program during the merge.	Report for investigation.
X1	Failure in internal communications between sections of program.	Report for investigation.
X6	Error in trying to establish a strategy for the merge.	Report for investigation.
XX	A non-specific error whilst trying to read or write a file.	Report for investigation.

Notes

1 Type J errors are followed by a message giving the file name of the file that the program was attempting to open.

2 Type L errors are followed by a message giving the file name upon which the transfer error occurred.

3 RESTART POINT 29. Error types R, S, T and Z are produced by the core resident routines, the remainder are produced by the sort proper, XSXA. After any error in #XSXA, the COBOL program may be reloaded by typing the message GO #*cccc* 29 (where *cccc* is the name of the COBOL program). Control is returned to the COBOL program accordingly. In this case, word 6 of *record-name* will be set to 1. In all other cases this word will be set to zero.

SORT END MESSAGES (DIRECT ACCESS)

At the end of the sort, three DISPLAY messages will be output as follows:

DISPLAY:- OUTPUT FILE SIZE

DISPLAY:- *X* RECORDS where *X* is the number of records output.

DISPLAY:- *Y* BUCKETS where *Y* is the number of buckets used in the output file.

The last message is not output if there is an output procedure.

The *LOAD steering line

The *LOAD steering line eliminates the need for the operator to FInd an object program on magnetic tape at run-time.

When *LOAD is not used consolidation terminates with the console message HALTED-CONSOLIDATED #*cccc*. To load the program the operator types FI #ABCD (see page 247) and when the program is loaded Executive replies HALTED LD.

If *LOAD is used, the messages

DISPLAY-CONSOLIDATED #*cccc*

DELETED-LO #*cccc* ON *nn*

are displayed, the program is automatically loaded and then the Executive message HALTED LD is displayed, as before. It may then be entered by typing GO #*cccc*20 (see page 247).

The format of the *LOAD Steering line is given on page 190.

ABORTIVE OBJECT RUNS

If the object program reads invalid input, e.g. punch errors on cards or paper tape, or there are hardware faults such as parity failures on magnetic tape or printer transfers, the program is suspended with a FAIL message on the console. This is the result of action, not by compiler-generated code, but by Executive. Full details can be found in the PLAN Reference Manual.

A further example of an object program 'going illegal' may be met if a subscript goes out of range. If there is any danger of this occurring, the user should insert a test of the value of the data name before using it as a subscript. Note that the value zero is not a valid subscript.

RUN-TIME SUBROUTINES, ERROR MESSAGES AND HALTS

As with other library routines the compiler incorporates in the object program (at Consolidation stage) only those routines required by the COBOL source: once only, and in the permanent part of the object program if the COBOL source is divided into overlay segments. Most of the routines are concerned with errors in the way peripherals have been handled, data errors or hardware failures, and will bring the object program to a halt with a SUSTY, no restart being possible. A complete list of these routines follows.

Note: In the following list, 'NAME' in the expression #NAME represents the four-character name of the COBOL source program.

Routine name	When included in object program	Size in object program	Effect at run-time
COBA	Incorporated (for running on 1904 upwards) when divisions are to be performed with both operands exceeding 23 bits or 6 decimal digits.	Lower Preset – 1 Program – 54	No halt at run-time.
COBB	As COBA, but incorporated in object program to be run on 1903 downwards.	Lower Preset – 1 Program – 43	No halt at run-time.
COBF	A call to this subroutine is produced for IF [NOT] NUMERIC statements when they refer to sterling fields.	Lower Preset – 3 Program – 82	The value in a sterling field i.e. 'DISPLAY-1/2/4/5' is examined to ensure that it is consistent with its Picture.
COBI	A call to this routine is generated by all READS of files assigned to magnetic tape.	Literals – 1 Program – 10 Upper Preset – 8	The routine is called in the exception condition coding when a block larger than the area specified in the BLOCK CONTAINS clause in the Data Division is read. The following message is typed 'LONG BLOCK DETECTED ON FILE nn' (see page 247).

Routine name	When included in object program	Size in object program	Effect at run-time
COBJ	Replaces own coding when the compiler limit for ENTERed subroutines has been exceeded.	Lower Preset – 1 Program – 7 Upper Preset – 10	The subroutine displays the message 'DUMMY ROUTINE ENTERED IN SENT nnnnnn' (see page 248).
COBL	Called by all READ, WRITE OPEN and CLOSE statements.	Lower Preset – 10 Program – 101 Upper Preset – 34	The subroutine checks that a particular COBOL input/output statement is valid at run-time, e.g. in the case of an OPEN, that the file is not already open and has not been closed 'with LOCK,' or in the case of a WRITE that the file is not open for input. There is a different entry point for each type of statement. If any check fails, an error message is typed and there is no restart. All the messages begin: 'SENT nnnnnn FILE nn' where nnnnnn = Sentence number of offending COBOL statement. and nn = File number. The remainder of the message is an indication of the error that has occurred, see page 249.
COBM	A call to this routine is generated for OPEN statements when the GENERATION NO. option appears in the COBOL File Description of the file being opened.	Lower Preset – 1 Program – 11 Upper Preset – 9	The subroutine is only entered if the generation number has not already been supplied by means of the ACCEPT FILE KEYS statement. The subroutine types the message: 'SET FILE nn GENERATION NO. IN WORD 0' (see page 247).
COBN	If there is a RERUN statement this subroutine is called to initiate dumping.	(not available)	The subroutine performs the MTH exception procedures for an output file controlling dumping at EOR. If there has been an unclearable parity error, 'CY' is typed and no restart is possible.

Routine name	When included in object program	Size in object program	Effect at run-time
COBN (continued)			If the file receiving dumping is one of the user's files COBL is called to ensure that the file is open for output. (If COBL types an error message in this case the 'sentence number' will be the characters RERUN). The buffers of all output magnetic tape files are cleared, and a dump is initiated using HDMT and HDUMP. The dump package will display messages listing which peripherals have been dumped, and all slow peripherals will be disengaged. Finally the subroutine types the message: ATTENTION TO STATE OF SLOW PERIPHERALS (See page 248). At this point the operator records the state of paper tapes, the last card read etc., and restarts the program by typing 'GO #NAME'. When the program is reloaded from a dump it will recommence at the instruction which types the message 'ATTENTION TO...'
COBO	Incorporated in object program by STOP RUN.	Literals – 1 Program – 19 Upper Preset – 7	Called at end of run to ensure that all files are closed. For any that are open the message 'FILE nn NOT CLOSED AT EOR' is displayed but there is no halt. (See page 247).
COBP	This subroutine is present if there is a GO TO ⊡ statement in the program.	Literals – 1 Program – 7 Upper Preset – 12	In a GO TO ⊡ statement (i.e. no destination paragraph) the code produced by the compiler is a CALL of COBP. The statement should be ALTERed before it is obeyed, but if it is not, COBP is entered and the message 'ATTEMPT TO OBEY AN UNALTERED 'GO' SENT nnnnnn' is typed. (nnnnnn = sentence number of GO TO ⊡ statement.) There is no restart.
COBQ	Called for WRITE statements to the printer when the AFTER option is used or there is no ADVANCING clause.	Program – 31	This subroutine prints a line after advancing the number of lines specified in the AFTER clause plus those specified by the previous WRITE statement of this file if it required printing BEFORE advancing. If possible the spacing requirements are combined, but the net effect of the advancing will always be as if the advancing had been carried out

Routine name	When included in object program	Size in object program	Effect at run-time
COBQ (continued)			by the WRITE statement which initially specified the advance.
COBR	Called for WRITE statements to the printer where the BEFORE option is used.	Program – 18	This subroutine prints a line after advancing the number of lines specified by the previous WRITE statement of this file if it required printing BEFORE advancing (there may, in fact, be no outstanding spacing to do). It will then store the spacing specified by the current WRITE statement in the Run Time File table. It will be taken from here on the next WRITE. Any spacing requirement outstanding when the file is closed will not be done (unless the file is subsequently re-opened).
COBS	Called by all OPEN statements referring to slow peripherals, by ACCEPT FROM, DISPLAY UPON, and ACCEPT FILE KEYS	Lower Preset – 6 Literals – 1 Program – 52 Upper Preset – 21	Used to allocate character peripherals to the object program. An ALLOT instruction is obeyed and if the peripheral is allocated (or already allocated) then one of the following messages is typed and the program suspended. LOAD FILE nn ON aa (see page 247.) LOAD ACCEPT DATA ON aa (see page 249). LOAD FILE KEYS ON aa (see page 249). There is no message for DISPLAY UPON. If the required peripheral is not available one of the following messages is typed and the program suspended. ALLOCATE qq TYPE ttt ALLOCATE qq (See page 249). On restarting the ALLOT instruction will be obeyed again and if the peripheral is allocated the LOAD... message typed. If the peripheral is still not allocated (because, e.g. it is the wrong TYPE) then the ALLOCATE message will be typed again.
COBZ	Present if COBR or COBQ are present.	Program – 15	Used to perform function common to both COBR and COBQ.

SORTING IN THE OBJECT PROGRAM

Sorting can be achieved either by using the SORT verb or by using ENTERed subroutines. The SORT verb uses a system based on a set of core resident routines and a separate sort program. Use of this system by ENTERing subroutines is described in parts of Chapter 13. References in the notes below are to those parts of Chapter 13.

Summary of SORT verb features

Files on magnetic tape, E.D.S., Twin-E.D.S., or F.D.S. may be sorted. Apart from the library file containing the sort program, all the files associated with the sort must be on the same device type. This device type is established by the SELECT . . . ASSIGN . . . clause for the sort-file.

Descriptions of input, output and work files will be found in Chapter 13. A direct access sort requires two workfiles. These must be named files, and they must have the same bucket size. If an output file is present, it must have the same bucket size as the workfiles. A magnetic tape sort uses from three to ten work tapes as specified by the user. The system is designed so that the number of decks in use during the sort does not exceed the number of work tapes used.

Keys are described in Chapter 13. The maximum number of keys is ten.

The names of the relevant core resident routines and sort program are given in Chapter 13. Use of the SORT verb causes the incorporation of the run-time routine CQ1R1S1. Run-time routine CQ2R2S2 will also be included if there is an output procedure.

Details of error messages, restart point 29 and end messages, are given above.

The notes given in Chapter 13 on file unit numbers, core store, workfile sizes, file extensions and console messages apply to the SORT verb.

The Procedure Division may contain more than one sort, but there may not be both a magnetic tape and a direct access sort.

Chapter 18 Own code, library and COBOL subroutines

COBOL SUBROUTINES

The end product of a compilation of a COBOL subroutine is a segment of semi-compiled program without the 'force-in bit' set. There is no logical difference at consolidation time between COBOL and other language subroutines. Hence the rules given on pages 257 to 259 apply equally to both types. When compiling a COBOL subroutine, the *CONSOLIDATE steering line should, of course, be omitted.

COBOL subroutines may be included in other language programs, and not only in COBOL programs. The calling sequence from a PLAN segment is as follows:

```
        CALL  1  subroutine-name
        LDX   3  word containing address of parameter-1
or      LDN   3  address of parameter-1
        LDX   3  word containing address of parameter-2
or      LDN   3  address of parameter-2
          .              .
          .              .
          .              .
          .              .
```

Accumulator 2 may be used as a modifier in the calling sequence, for either form of parameter load. COBOL subroutines preserve all accumulators.

Users may be aware that the parameters of subroutines need not always be in the form

```
        LDX   3  word containing address
```

but may be in the form

```
        CALL  0  address-calculation-subroutine
```

(see, for example, subscripted parameters on page 83). Parameters of COBOL subroutines may also be in this form. There is little point, however, in them being so, since the subroutine evaluates the parameters only *once* on entry and not every time it wishes to use them.

All data in COBOL subroutines is private, that is, there is no common data in COBOL subroutines. Hence all data areas are included in the overlay unit in which the subroutine is included. This means that both the program and data areas of a subroutine are loaded in their initial state whenever the overlay in which the subroutine is included is reloaded.

Example of a COBOL subroutine

The subroutine SUBR examines each character of a field, compares it with '9' and replaces characters not equal to '9' by spaces. It is called by a COBOL program. The entries in the Data and Procedure Divisions necessary to call the subroutine are as follows:

```
DATA DIVISION.

.
.
.

01 A PIC X(120).
```

```
PROCEDURE DIVISION.
        .
        .
        .
        .
CALL SUBR USING A.
        .
        .
        .
        .
```

The subroutine must be written as follows. Note that the various division, section and paragraph headers must be written although for the Identification and Environment Divisions the only information which must be included under them is the subroutine name in the PROGRAM-ID paragraph and the object computer. The latter need not be in the same as that specified for the main program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBR.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER. ICT-1902.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
    77 COUNT PIC 9(4) VALUE 1.
LINKAGE SECTION.
    01 A.
    02 B PIC X OCCURS 120.
PROCEDURE DIVISION.
PARAGRAPH-1.
    IF COUNT = 121 EXIT PROGRAM.
    IF B(COUNT) NOT = 9 MOVE SPACES TO B(COUNT).
    ADD 1 TO COUNT.
    GO TO PARAGRAPH-1.
```

Further information on the language aspects of COBOL subroutines can be found on pages 19 (Identification Division), 41 (Linkage Section), 65 (Procedure Division), 76 (CALL) and 87 (EXIT).

CONSOLIDATION OF COBOL PROGRAMS AND INCLUSION OF COBOL OR OTHER LANGUAGE SUBROUTINES

There are two methods of consolidating COBOL programs: in the first the Consolidate run is part of the COBOL compilation; in the second the Consolidator used is an independent program.

The chief difference between the two is that in the first it is not possible to include subroutines in an overlay unit; all subroutines go into permanent program.

Consolidation as part of the COBOL compiler

User's routines (Own Code specified in the COBOL program by ENTER verb) and subroutines on a library tape are incorporated in the object program at Consolidate time in the same way that other external routines (e.g. those of the Magnetic Tape Housekeeping package) are incorporated.

The sequence of events at Consolidate time, when such routines are to be incorporated, is:

1 The Consolidator reads the segment leaders of the semi-compiled object program; these may be either compiler-generated (in store), or on magnetic tape, paper tape or cards.

2 If there are unsatisfied cues, (that is to say references to apparent program labels which have not been defined as COBOL procedure names and which may be the Own Code routine names quoted in ENTER statements) the Consolidator then reads the peripheral specified in the *SUBROUTINES Steering line. (As stated on page 186 this Steering line must always be used when compiling a COBOL program with one or more ENTER statements, or when the library tape is not the same as the compiler tape, i.e. when the standard routines used by the compiler are not on the compiler tape itself.)

3 If any of the unsatisfied cues match any of the names of routines read via the specified peripheral, these routines are then copied on to the end of the semi-compiled program, on whichever peripheral it may be.

4 If there are still unsatisfied cues, (which may refer to Magnetic Tape Housekeeping routines, standard run-time subroutines, e.g. COBO, the overlay package or library routines referenced in ENTER statements) the Consolidator reads the compiler tape, and as in stage 3, if it finds matching routine names, it adds these routines to the semi-compiled program.

All routines incorporated in stages 3 and 4 above go into the permanent part of the object program, if the COBOL source was divided into overlay segments.

It should be noted that at present Own Code is only acceptable in semi-compiled form, that is, as the product of, for example, a PLAN or FORTRAN compilation. Furthermore it must conform to certain requirements, which are set out below.

DEFINITIONS

A 'file' of semi-compiled to be consolidated into the object program consists of *segments* which in turn comprise *records.*

Record

An acceptable *record* of semi-compiled contains a binary count of the number of words in the record in the second character of the first word of the record (maximum value 18). There is one record per 80-column card on punched cards, one per block terminated by newline on paper tape, and a variable but whole number of records per block on magnetic tape. If there is only one record per block on magnetic tape there may be, and if there is more than one there must be, a record whose first character is #07 preceding the first record of normal semi-compiled at the beginning of each block. The maximum size of such a block is 128 words.

Consolidated *or* unconsolidated semi-compiled generated direct by a compiler or by a standard I.C.T. Library Update program can be used *provided that* the second character of each record contains the binary word count. The following current I.C.T. 1900 languages e.g. obey this convention: COBOL, NICOL, PLAN 2, PLAN 3, FORTRAN 4, EMA.

PLAN 1 does not obey it and cannot be used for this purpose.

Segment

A logical unit of semi-compiled commencing with a record containing #03 as its first character and terminated by a record containing #04 as its first character.

File

All physical data recorded in the appropriate medium between the header and terminator, exclusive of both. It consists in general of *segments* and 'extraneous data'. The following 'extraneous data' only may be present in such a file and will be ignored by the COBOL Consolidator:

Binary programs (e.g. loaders or complete programs)

Pause Blocks (a record commencing with #13)

and on magnetic tape:

Control information produced by the library routine #XPMU – e.g. blocks beginning #72, and Request Slips (Tapemark followed by a block beginning #73nn0001)

Control information produced by compilers, – Start-of-Subfile Sentinels (Tapemark followed by #00000006)

End-of-Subfile Sentinels (Tapemark followed by #40000000)

Note: A file of consolidated semi-compiled will normally consist of binary program (a loader) which will be ignored by the COBOL Consolidator, and semi-compiled segments, which will be recognized and if necessary incorporated into the object program.

FORMATS

File format on paper tape

One or more reels of paper tape with no special headers. An intermediate reel is terminated by a Pause Block, and the final reel by a block containing ****newline. If there is no such terminal block the operator will observe, on FIXing the unit (as instructed by each Pause Block), that there is no more data to be read, and will then re-enter the compiler by GO 28.

File format on 80-column cards

A pack of cards with no special header and terminated by a card containing **** in columns 1 to 4. If there is no such terminating card the operator observes, on FIXing the card reader, that no more data is to be read, and types GO 28.

File format on industry compatible magnetic tape

One reel of magnetic tape with a standard I.C.T. 1900 Header Label (see System Manual), terminated by an End-of-Composite-File Trailer Label (see System Manual). For the time being an alternative terminator of two consecutive Tapemarks is also acceptable.

THE FORCE-IN' BIT

It is important to realize that if the 'force-in bit' is set for any segment of semi-compiled offered to the COBOL Consolidator, that segment will be incorporated into the object program, whether or not it is wanted.

The presence of the 'force-in bit' may be recognized as follows:

A segment is introduced by a record beginning with #03. Starting at the third character of the first word of this record, take the first character (say 'X') with a value *not* in the range #00 to #07. If 'X' is in the first word of the record the 'force-in bit' for this segment is *unset*.

If 'X' is in the second or a subsequent word, then the fifth bit of the third character before 'X' is the 'force-in bit', and if its value is 1 it is set, if 0 unset for this segment.

If the 'force-in bit', recognized as above, is set for a segment, then this segment will be incorporated into the COBOL object program in all circumstances *unless* a segment with the same name, or a cue-within-segment of the same name, has already been met.

The details of how the 'force-in bit' is set in languages other than COBOL are available in the appropriate literature. In current versions of PLAN, for example, the PLAN Compiler sets the bit if the segment is given a program name in columns 16 and following of the first line of the segment: otherwise it is not set.

The COBOL Compiler sets the bit in all semi-compiled segments of object code; it is not set in segments of 'run-time routines' and routines of the Magnetic Tape Housekeeping package on the COBOL system tape.

THE PAUSE BLOCK

As indicated above a Pause Block (on paper tape only) always suspends Consolidation with a FIX message. The decision as to whether it means 'end of subroutine file' is passed to the operator, who will see whether all the data supplied has been read on re-allocating the peripheral. It is recommended, therefore, that where possible (in the interests of speed) all Pause Blocks in a file of semi-compiled on paper tape should be removed before offering it to the COBOL Consolidator, and a terminator record of ****newline spliced on to the end of it.

ORDER OF SUBROUTINES

The COBOL Consolidate phase scans serially *once* forward through the file specified in the *SUBROUTINES Steering line (if any), accepting ENTERed routines, routines which they themselves CALL and so on to any level. It is therefore essential that any 'lower level' routine (i.e. one not referenced explicitly or implicitly by the compiler-generated object code) should *follow* its calling routine on the file.

After scanning the *SUBROUTINE file (if any) and the COBOL system tape the Consolidation process ceases *either* when the system file terminator is met, *or* when all routines ENTERed by the COBOL source (and associated CALLed routines) have been found and incorporated into the object program.

Independent consolidation

For independent Consolidation with the master program and subroutines on magnetic tape or slow peripherals, the free-standing Consolidator XPC1 should be used. This program is described in the 1900 Series Library Specifications.

The file containing the semi-compiled master program (i.e. that COBOL program which had no *LINKAGE steering line) should be identified by the *INIS directive to the Consolidator: this file contains the steering segment for the whole program. Subroutines, of whatever language, may be

1 in the user's other input file, or

2 in his slow peripheral library, or

3 on his library tape.

If the user's master program had no *OVERLAY steering line, all subroutines will be included in permanent program. If this steering line was present, then

1 all standard run-time routines will be included in permanent program.

2 all CALLed or ENTERED routines will be included in permanent program if they are referenced in more than one overlay unit of the master program. Otherwise, they will be included in the overlay unit in which they are called.

3 subroutines may call other subroutines. These lower-level subroutines will be included in permanent program.

Chapter 19 Testing and correcting

MONITORING THE SOURCE PROGRAM

A COBOL source program may be effectively monitored at object run-time by inserting DISPLAY and WRITE statements at strategic points in the source program, e.g. at the ends of sections or in program loops. Thus, the contents of any areas defined in the Data Division may either be displayed on the console typewriter or be written by the line printer. In the latter case, the user should either define an extra record in a file ASSIGNed to the printer or if there is no such file, define one for testing and ASSIGN it to the printer. Data fields are then MOVEd to this area for monitoring.

The user may if he wishes evolve more elaborate systems of monitoring. For example, he might insert a count in his program, update and test it at the end of a loop, and DISPLAY or WRITE the contents of the areas in which he is interested when the count reaches a certain value, i.e. when the loop has been performed a certain number of times. The value itself might be defined as a constant, which can be changed for different compilations and object runs, in WORKING-STORAGE. Since all these 'testing only' insertions will have to be taken out of the source program before final compilation, it will be best to put them in one or more separate paragraphs or sections of their own; preferably at the end of the program, which can then be PERFORMed during testing, e.g.

```
                    (program)
                    PERFORM MONITOR-PRINT-A.
                    .....
                    .....
                    .....
MONITOR-PRINT-A.
                    ADD 1 TO TEST-COUNT-A.
                    IF TEST-COUNT-A EQUAL TO 50 DISPLAY 50, OLD-TOTAL,
                    NEW-TOTAL, QUANTITY.
(paragraph)
                                        (etc.)
```

The user is reminded that although one DISPLAY statement may be used to display the contents of a number of areas, as above, the total size of all the areas and literals must not exceed forty characters. In some cases it may be better to have separate DISPLAY statements for each field, so that they may be easily taken out of the source program when it is required to vary the fields that are displayed, for different compilations and object runs.

Another means of achieving this effect is to insert a card or paper tape READ statement for testing purposes. The card or block read would then be punched with parameter values indicating which areas are to be monitored. In this case, a special file would usually have to be defined and ASSIGNed to the chosen input device.

Suppose, while debugging, the programmer is interested in the contents of six areas (AREA-1, AREA-2 etc.) but wishes to display different combinations of them on different occasions. A file might be named TEST-FILE, ASSIGNed to card reader and be given the structure:

```
01 TEST-PARAMETER.
02 CODE PICTURE X OCCURS 6.
02 FILLER PICTURE 9(74).
```

At the beginning of the Procedure Division, OPEN INPUT TEST-FILE READ TEST-FILE.... would be inserted; at the end of the loops under test, conditional statements of the form:

IF CODE (n) NOT EQUAL TO 0 PERFORM TEST-PRINT-n.

(where n has one of the values 1 to 6)

and in a separate TEST-SECTION, paragraphs of the form:

TEST-PRINT-1.

DISPLAY AREA-1.

TEST-PRINT-2.

DISPLAY AREA-2. (etc.)

By punching non-zeros in the appropriate places in the first six columns of the parameter card, or leaving them blank, the user could thus get any required combination of the six areas displayed at run-time. This system could of course easily be combined with the first described.

It may be noticed here that the user can achieve the same effect by assigning names to the hardware switches of the 1900 Series in the SPECIAL-NAMES section of the Environment Division, and testing them in the Procedure Division.

Statements and tests added to the source program for debugging purposes, as above, may be either

(a) manually inserted in and extracted from a source program on cards or paper tape before compilation, or

(b) both inserted and extracted as Correction lines in a *CORRECT run (see page 188 and below) when the source program is on magnetic tape.

CORRECTING THE SOURCE PROGRAM

Corrections may be applied either to the original COBOL program or to the COBOL version translated from a Rapidwrite source. They cannot be applied to a Rapidwrite source using the *CORRECT Steering line.

By corrections is meant the deletion, insertion or replacement of lines in the COBOL source program and/or the transcription of the source program from one medium to another. The lines to be corrected must be in the appropriate format, either on cards, paper tape or magnetic tape. The *CORRECT Steering line and its associated minor directives (see page 188) are used to tell the compiler from which input medium the Correction lines are to be read, as well as the input medium of the source program and the output medium of the corrected source. The formats of the three types of Correction lines are given below.

Deletions

To delete one line from the source program a card is punched (or the same image is punched as a paper tape block followed by newline) with the sequence number of the original line in the first six columns followed by a space in column 7, the word DELETE in columns 8 to 13, and a space in column 14.

To delete a group of lines the punching is:

Columns	Punching
1 to 6	sequence number of the first of the group of lines
7	space
8 to 13	DELETE
14	space
15 to 18	THRU
19	space
20 to 25	sequence number of the last of the group of lines to be deleted.

Notes:

1 The first sequence number must always be less than the second.

2 Unnumbered lines following the last deleted line are not affected.

Insertions

A single line may be inserted in the source program by punching it on a card or paper tape in standard COBOL format and assigning it a unique sequence number, which determines where it will appear in the corrected source.

A group of consecutive lines may be similarly inserted by assigning a unique sequence number to the first of the lines. Sequence numbers for the subsequent lines may be omitted, in which case, the compiler assigns numbers to them, incrementing by 100. If the last number assigned by the compiler is equal to or greater than that of the original source line following the insertion point, the compiler will renumber these following source lines, but only as far as necessary.

Replacements

A line may be replaced in the source program by a Correction line in the same format and with the same sequence number.

A single line may be replaced by a group of lines by assigning the first of the group the same sequence number as the original line. Sequence numbers for subsequent lines in the replacement group may be omitted, in which case the compiler proceeds as described in the second paragraph of the section on *Insertions* above.

Notes:

1 The last of the Correction cards or blocks, whether the user is deleting, inserting or replacing, must be followed by an End Marker card or block with **** in the first four character positions (followed by newline if paper tape) unless *AMENDMENTS NONE is present in the Steering line.

2 If the user inserts a card or block with the word CORRECTIONS in the first eleven character positions before the first of the Correction cards or blocks, it will be recognized by the compiler and ignored.

3 The Correction phase of the compiler (called for by *CORRECT Steering line) treats source program sequence numbers a little differently from the way they are treated by the Compile phase (called for by *COMPILE). For the latter see page 189.

A source program input to the Correction phase, whether on cards, paper tape or magnetic tape, may have sequence numbers on all lines, on none or only on some. Resequencing will only be done by the Correction phase where necessary, and as far as necessary. Thus unsequenced lines will be assigned numbers in increments of 100, and if sequenced lines are then met in the source program they will only be resequenced if the compiler-assigned numbers require it, as in the following example:

```
000700  MOVE.....
        ADD.......
        ADD.......
        GO TO...
000800  MOVE.....
```

The second, third and fourth lines would be given the numbers 800, 900 and 1000 in the new corrected source program, the fifth line renumbered 1100, and following lines also resequenced if necessary.

If the fifth line in the input source program had been numbered 1100 (or more) it would not have been resequenced.

Note: Sequence errors in the source program are never flagged by the Correction phase, but sequence errors in the Correction lines are flagged (message number 607).

Clearly it will usually be impossible – depending on the position of the Correction lines – to insert Corrections in a source program which contains sequence errors. Such a program must first be put through the Correction phase without any Correction lines and then the new source, with valid sequencing, can be again put through the Correction phase this time with Correction lines referring to the appropriate source lines.

An input source program on magnetic tape will normally itself be the output of a previous *CORRECT run, and all its lines will therefore have been sequenced, if necessary by the Correction Phase.

4 The order of input for a *CORRECT run when all parameters are to be read from the same peripheral (i.e. from cards or paper tape) is:

Steering lines

End marker (****)

Correction lines

End marker (****)

COBOL source program

End marker (****)

5 The same line cannot be both DELETEd and REPLACEd.

Chapter 20 Compiler limits

The following error messages, already quoted on pages 201 to 233, signal to the user that various limits have been exceeded:

013, 100, 101, 103, 114, 138, 139, 175, 179, 195

200, 207, 210, 413, 506, 508, 620, 628, 635, 644, 645,

650, 700, 714, 738, 739, A5, A8

The limits are imposed either by hardware, the amount of core store made available to the compiler, or by software, the limits set by the compiler on the sizes of work areas, tables, stacks and so on. The latter are flexible; so far as possible the compiler will set different limits for each table and stack according to the core availability (and complexity of source program) for each separate compilation run. They are therefore so variable that it is impossible to give exact figures, but the following are some additional notes on the significance of these messages and the action to be taken by users when limits are exceeded.

013 SENTENCE TOO LONG/COMPLEX

A variety of circumstances may produce this message, e.g. arithmetic statements with too many operands, MOVE statements with multiple target fields, ENTER or GO.....DEPENDING statements with a large number of operands (usually up to 18 are acceptable), deeply nested conditions, a sentence containing a large number of statements, etc. The message will quote the sequence number of the first line of the sentence, and the user should have no difficulty in deducing what is objectionable in the sentence. Remedies will vary: an ADD Statement with 18 operands can be changed to two ADD statements, each with only 9 operands, and the same applies to MOVE's with multiple receiving fields. If an ENTER Statement has too many parameters, an attempt can be made to *tabulate* some of them, i.e. make them consecutive in store, so that only one parameter, the start address of the table, need be passed on to the routine. Deeply nested conditions can be simplified by breaking them into a succession of conditions; these will generate very little more, if any, object code, and it will in addition be much easier to follow the logic of the COBOL statements. Sentences containing a large number of statements can be broken up into more sentences by inserting full stops.

100 COMPILER STORAGE EXCEEDED: PICTURE/VALUE IGNORED

A field description in the Data Division, usually a printed output field with complex editing, contains too many clauses or has too complex a Picture.

Remedy: split the field into subfields and reference the latter. For example, suppose there is a statement

MOVE (field) TO AVERAGE-COST (ITEM)

where AVERAGE-COST is defined as

02 AVERAGE-COST PIC £,ZZZBB99BB99V.9 OCCURS 5 ⊡

Suppose, too, that this Picture is found to be too complex for the core store available to compilation, and message 100 is flagged.

A possible remedy would be to subdivide AVERAGE-COST into:

03 AV-COST-1 PIC £,ZZZBB99 ⊡

03 AV-COST-2 PIC BB99V.9 ⊡

also subdivide the source field into 2 corresponding subfields and then MOVE these into AV-COST-1 and AV-COST-2 (subscripted by ITEM).

101 COMPILER STORAGE EXCEEDED: INCORRECT OBJECT CODE

Refers to compiler limit of 64 files in the Data Division. If this limit is exceeded a number of the files will almost certainly have been assigned to the same peripheral units, and if say, FILE-A and FILE-B are both assigned to CARD-READER 1, the usual method of processing will be to OPEN one of the files, READ it and CLOSE it before OPENing and processing the second file. In this situation the second file need only be defined as a record or records in the first file, which is one way of reducing the number of files. There should also never be any need to define more than one file assigned to the same Printer, only one file with a number of different records, plus, if necessary extra "record areas" in WORKING-STORAGE.

If the number of files still can not be kept within the limit by this method the user may either define and process some of them in ENTERed routines (but see note (i) on page 144), or as a last resort divide the source program into two or more separate programs. These programs can then be separately compiled as segments or "subroutines" and consolidated together into one program.

103 FIELD TOO LARGE

Refers to the compiler limits of 349,525 words for a field (almost certainly a table), or 262,243 OCCURrences of an item in a table. These are arbitrary limits; clearly the store size of most machines will effectively impose much lower ones. Tables exceeding these limits will have to be held on an external medium, magnetic tape, discs, etc., and read into store and used as required, in sections.

114 NOT ENOUGH LOWER

See page 243.

138 COMPILER STORAGE EXCEEDED: TOO MANY FILES

See note on message 101 above.

139 COMPILER STORAGE EXCEEDED: RECORD SIZE ALTERED

There are too many and/or complex fields in a record. The simplest action to avoid this is to define the record as two (or more) records, as in the following example. Suppose RECORD-A contains 120 fields

```
02 FIELD-1 ⊡
   03 FIELD-2 PIC X ⊡
   03 FIELD-3 PIC X ⊡
02 FIELD-4 PIC X ⊡
02 FIELD-5 PIC X ⊡
   (etc.)
     .
     .
     .
02 FIELD-120 PIC X ⊡
```

If message 139 is printed, the definition of RECORD-A should be replaced by, say, 2 records, thus:

```
01 RECORD-A-1 ⊡
   02 FIELD-1 ⊡
      03 FIELD-2 PIC X ⊡
```

```
        03 FIELD-3 PIC X ⊡
    02 FIELD-4 PIC X ⊡
    02 FIELD-5 PIC X ⊡
            .
            .
            .
    02 FIELD-60 PIC X ⊡
01 RECORD-A-2 ⊡
    02 FILLER PIC X OCCURS..... ⊡
```

(where the size of this FILLER is equivalent to the total sizes of FIELD-1, FIELD-2.....FIELD-60 above)

```
    02 FIELD-61 PIC X ⊡
    02 FIELD-62 PIC X ⊡
            .
            .
            . (etc.)
            .
            .
    02 FIELD-120 PIC X ⊡
```

This takes advantage of the fact that all records in a file share the same area.

If RECORD-A is in an output file or in Working-Storage the only change required in the Procedure Division is to substitute RECORD-A-2 for RECORD-A in WRITES, MOVES etc.

If it is in an input file a similar substitution should be made in MOVES or any other statements addressing the record level (but not in READS which reference the *file* name).

In this way the over-large record can be split up into as many pseudorecords as may be necessary to get within the compiler limit, defining a different part of the original record in each one, and defining the remainder of the pseudorecord as a FILLER.

Another way of overcoming the problem, of course, is to define the original record very simply, then define parts of it in full in different Working-Storage locations to which the parts are MOVEd, and in which they are then processed.

175 COMPILER LIMIT EXCEEDED

See note on message 013 on page 265.

179 TOO MANY SECTIONS

There are too many overlay segments in the source program, Remedy:

1 Reduce the number of segments: this will mean there will be fewer but larger segments, and it will probably mean that the size of the largest segment will increase.

2 If this satisfies the compiler limit on sections, but then makes the object program too large for available computer size, follow the procedures suggested under message 210.

195 TOO MANY PROCEDURE NAMES

Since the compiler calculates the size required for its tables (Names Tables, Paragraph Tables etc.)

during the initial scan of the Source program, this message, which is printed later on in compilation, should result from only one circumstance: the referencing in the Procedure Division of a large number of procedure names which are never *declared*. The compiler allocates extra table storage to allow for a certain number of undeclared paragraphs (the amount, being a percentage, will vary from compilation to compilation), but, if there are more than this number, message 195 is printed and compilation halts. Furthermore in the latter case the halt occurs *before* message 190 UNDECLARED PARAGRAPH is printed for each such paragraph, and the user must therefore desk-check his source program and correct his errors by declaring all the paragraphs referenced, before re-compiling. In view of the extra allowance made by the compiler this should only happen if there has been a very large number of errors in the writing or punching of the source program.

200 LOWER STORAGE EXCEEDED (BY nnnn WORDS)

See page 243.

207 ENTER: TOO MANY INCLUDED ROUTINES

The number of routines that may be ENTERed in a COBOL program depends chiefly on

(a) the amount of core available to compilation,

(b) the length of the routine names (the shorter the names, the more routines are acceptable).

If the limit is exceeded the user may adopt one of several remedies:

1 Make more store available to the compiler, if possible.

2 Try to reduce the number of routines by rewriting them in COBOL where possible, or by making some of them subroutines CALLed by other routines, and so not specifically ENTERed in the COBOL program.

3 Partition the source into two or more separate programs, each with a quota of ENTERed routines.

4 Partition the program into segments which may be recompiled separately but consolidated together.

210 OBJECT PROGRAM HAS EXCEEDED COMPUTER SIZE BY nnnn WORDS

Remedies:

1 Give the object program more store (in OBJECT-COMPUTER paragraph) if possible.

2 Segment the source program into overlays, if it is not already segmented, or if it is, attempt to reduce the size of the largest overlay, probably by creating more segments.

3 Investigate e.g. whether such COBOL space-saving options as SAME AREA, REDEFINES, RENAMES can be used; whether any reduction can be made in the number of files defined (see suggestions under message 101); whether the PERFORM verb has been used as widely as possible (see PERFORM on page 94); whether tables held on external media may be read in and used piecemeal without too great a loss in efficiency; and in general examine all areas of the program for optimization. The use of COMP SYNC RIGHT areas is particularly important in saving conversion coding.

In the last resort, of course, it may be necessary to partition the source program into a suite of two or more programs. See also notes on "COBOL subroutines" under message 207 and 101.

413 TOO MANY OPERANDS

Refers only to *arithmetic* statements (the maximum number of operands permitted at present being nine). It should in all cases be easy to expand the arithmetic statement with more than nine operands into two or more similar statements, achieving the same result

e.g. ADD A, B, C, D, E, F, G, H, I J, K, L, GIVING TOTAL ⊡ can become

ADD, A, B, C, D, E, F, GIVING TOTAL ⊡

ADD G, H, I, J, K, L TO TOTAL ⊡

Care will of course have to be taken over the SIZE ERROR clause, if present; e.g. the user may wish to ensure that *all* the operands have been totalled, even though both ADDs have a SIZE ERROR branch and an overflow actually occurs on the first ADD. In this event the SIZE ERROR branch of the first ADD would be a GO to the next ADD, which would have to be given a paragraph name.

506 SECTION EXCEEDS 32767 WORDS

See explanation against the message on page 223. The program must be segmented into overlays, or resegmented.

508 TOO MANY LABEL REFERENCES

This refers to the "cue table" of forward jumps which the compiler makes up. An entry would be made, for example, when the statement GO TO PARAGRAPH-X is met and PARAGRAPH-X has not been declared *before* the GO.

The user should reshuffle the order of procedural statements (i.e. the order of the paragraphs and sections) so as to reduce the number of these forward references until they are within the compiler limit.

620 O/P LOWER nnnnnn WORDS TOO LARGE

See page 202.

628 TOO MANY FILES IN STEERING LINES

The present limit is thirteen and in existing modes of compilation there should never be any circumstances in which the user needs or wishes to quote more than thirteen files in his Steering Lines. If the message is printed, therefore, it indicates a *user error*, and the Steering Lines should be examined for the superfluous files.

635 COMPILER LIMITS EXCEEDED

There are too many data and/or procedure names in the source program and the compiler-generated tables incorporating them will be too large for the store available to compilation. In reducing the number of names some of the suggestions in note 3 under message 210 may be useful. Otherwise a major rewrite may be necessary, or the source program may have to be partitioned into two or more separate programs.

Note: The length of the names is also important. It will help considerably if they can be shortened, since in the Names Table (which exists only in the Analysis Phase) the actual COBOL names are preserved.

If plenty of core is available for the object program, PERFORMed paragraphs can instead be inserted in-line where needed in the COBOL Source, thus reducing the number of paragraphs. Similar considerations apply to other paragraphs: a re-shuffling of the order of the COBOL procedures may eliminate GOs and the paragraph-names they reference.

Each FILLER counts as a data-name, although the size of its entry in the Names Table is much smaller than that of a genuine data-name. Thus FILLERS should be eliminated where possible. In the case of Print records this can sometimes be done by replacing FILLERS by "B's" prefixed or suffixed to the Pictures of the fields following or preceding the FILLER.

644 COMPILER LIMITS EXCEEDED

See note under message 508 on page 223.

645 COMPUTER SIZE EXCEEDED

This message is printed during consolidation if the core storage required by the object program exceeds the figure quoted in the OBJECT-COMPUTER clause. *Either* a larger figure must be quoted in the latter clause, if possible, *or* the program must be segmented into overlays or amendments made to cut down its size (for which see suggestions under messages 210 and 635).

650 COMPILER NEEDS MORE MAGNETIC TAPES

Self-explanatory: in fact this message should never occur in current versions of the compiler since four decks are sufficient for all types of compilation, with dismounts/mounts where necessary. The message *may* legitimately occur in later versions if extra decks are needed for e.g. the implementation of COBOL "COPY" and related facilities.

700 COMPILER STORAGE EXCEEDED: PICTURE/VALUE IGNORED

See note under message 100 on page 210.

714 NOT ENOUGH LOWER

See note under message 114 on page 211.

738 COMPILER STORAGE EXCEEDED: TOO MANY FILES

Means *either* that there are more than sixtyfour files in the source program, in which case see note on message 101 on page 210, or that there are too many and/or complex fields in a record in which case see note on message 139 on page 212.

739 TOO MANY FILES

See note on message 101 on page 210.

A5 TOO MANY PROCEDURE NAMES

If this message is printed it almost certainly results from a compiler error or restriction. It is no longer a valid message.

A8 TOO MANY NAMES

This too is no longer a valid message and can only result from a compiler error or restriction. At present it will occur if the word "PROCEDURE" is used in comments in the Identification Division.

PART 4 APPENDIXES

Appendix 1 Alphanumeric data character set

0 to 9		(	left parenthesis	
:	colon	)	right parenthesis	
;	semicolon	*	asterisk	
<	less than	+	plus sign	
=	equal sign	,	comma	
>	greater than	-	minus	
?	question mark	.	full stop or decimal point	
	space	/	oblique stroke	
!	exclamation mark	@	at sign	
"	quotation mark	A to Z		
#	number sign	[	left bracket	
£	pound sign	$	dollar sign	See note
%	percentage	]	right bracket	
&	ampersand	↑	upwards arrow	
'	apostrophe	←	leftwards arrow	

The way in which this character set is restricted for alphabetic fields or numeric fields with USAGE DISPLAY-n is described under USAGE (see page 59).

Note: Data input to object programs on paper tape should not contain these characters. Object programs read paper tape in 'normal' mode and in this mode characters are read into store as two characters.

Appendix 2 Reserved words

In the following list, optional words are marked with a single dagger [†]. Words which are sometimes optional and sometimes key words depending on the context are marked with a double dagger [††] (see page 16).

All the other members of the list are COBOL keywords.

† ABOUT
ACCEPT
ACCESS
ACTIVE-TIME
ACTUAL
ADD
† ADVANCING
AFTER
ALL
ALPHABETIC
ALPHANUMERIC
ALTER
† ALTERNATE
AN
AND
APPLY
† ARE
† AREA
† AREAS
ASCENDING
ASSIGN
AT
† AUTHOR
BEFORE
BIT-nn
(where 00≤nn≤23)
BITS
BLANK
BLOCK
BY
CALL
CARD-PUNCH
CARD-READER
CASSETTES
CASSETTE-*
CHANNEL-1
CHANNEL-2
CHANNEL-3
CHANNEL-4
CHANNEL-5
CHANNEL-6
CHANNEL-7
† CHARACTERS
CHECK
CHEQUE
† CLASS
CLOSE
COMMA
COMP
COMPUTATIONAL
COMP-1
COMPUTATIONAL-1
CONFIGURATION
CONSTANT
† CONTAINS
† CONTROL
COPY
CORE-MT
CORE-EDS
CORRESPONDING
CORR
† CURRENCY
DATA
*DATE
† DATE-COMPILED
† DATE-WRITTEN
DECIMAL-POINT
DELETE
DEPENDING
DESCENDING
DIGITS
DIRECT
DISPLAY
DISPLAY-1
DISPLAY-2
DISPLAY-3
DISPLAY-4
DISPLAY-5
DIVIDE
DIVISION
EDS
EDS-*
ELSE
†† END
ENTER
ENVIRONMENT
EQUAL
EQUALS
ERROR
† EVERY
EXAMINE
EXCEEDS
EXIT
FD
FDS
FDS-*
FILE
FILE-CONTROL
FILE-MESSAGE
FILLER
FIRST
FIRST-LEVEL
FLOAT
† FOR
FROM
GENERATION-NO
GIVING
GO
GREATER
HAND-KEYS
HIGH-VALUE
HIGH-VALUES
ID
IDENTIFICATION
IF
IN
INCLUDE
INDEX-BUFFER
INDEXED
INPUT
INPUT-OUTPUT
† INSTALLATION
INTO
INVALID
I-O
I-O-CONTROL
†† IS
JUST

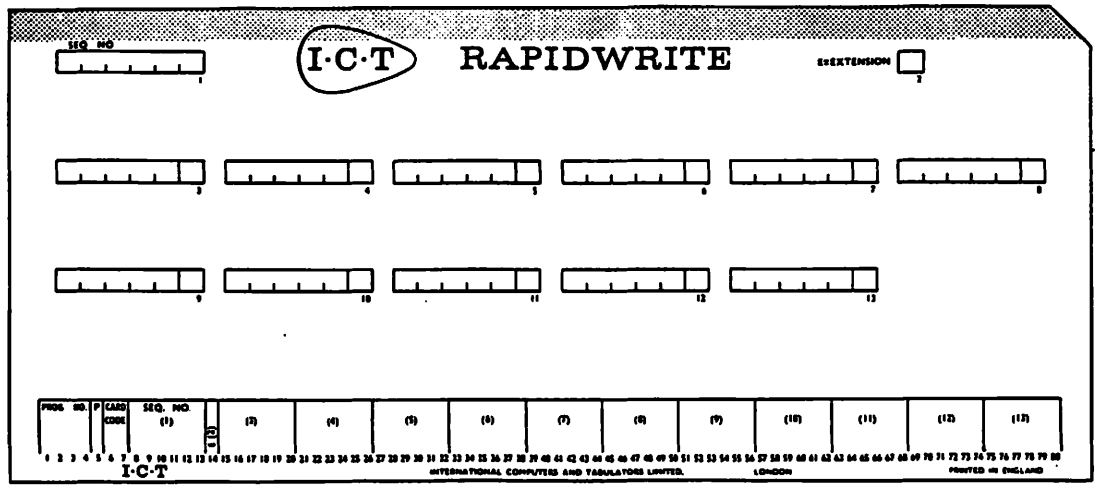
JUSTIFIED
KEY
KEYS
LABEL
LEADING
LEAVING
LEFT
LESS
† LINES
LINKAGE
† LOCATION
LOCK
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS
MCF
MCF-*
MEMORY
† MODE
MOVE
MULTIPLY
† MULTIPLE
NEGATIVE
NEXT
NO
NOT
NOTE
NUMERIC
OBJECT-COMPUTER
OBJECT-PROGRAM
OCCURS
†† OF
OFF
OMITTED
†† ON
OPEN
OR
ORGANISATION
ORGANIZATION
OTHERWISE
OUTPUT
PAPER-PUNCH
PAPER-READER
PERFORM
PIC
PICTURE
† PLACES
POINT
POSITIVE
PRINTER
PROCEDURE
PROCEED
PROCESSING
PROGRAM
PROGRAM-ID
† PROTECT
QUOTE
QUOTES
RANDOM
RANGE
READ
†† RECORD
RECORDING
RECORDS
REDEFINES
REEL
RELEASE
† REMARKS
RENAMES
REPLACING
RERUN
RESERVE
RETURN
REWIND
REWRITE
RIGHT
ROUNDED
RUN
SAME
SECOND-LEVEL
SECTION
† SECURITY
SEEK
SEGMENT-LIMIT
SELECT
SENTENCE
SENTINEL-PROCESSING
SEQUENCED
SEQUENTIAL
SIGN
SIGNED
SIZE
SORT
SORT-*
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
† STATUS
STOP
SUBTRACT
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONISED
SYNCHRONIZED
TALLYING
TAPES
TAPE-*
† THAN
† THEN
THROUGH
THRU
*TIME
†† TIMES
†† TO
TRANSFER-REPLY
TYPE
TYPEWRITER
UNEQUAL
UNTIL
UPON
UPPER-BOUND
UPPER-BOUNDS
† USAGE
USING
VALUE
VALUES
VARYING
† WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

Appendix 3 Facsimiles of Rapidwrite forms and card

In Rapidwrite there is no separate form for the Identification Division; the program name is written at the head of the Environment Division form.

Note also that some headings on the Environment Division form are not applicable to the 1900 Rapidwrite defined in the current manual.

The Procedure Division card illustrated below is a general card only; some of the cards used in Rapidwrite may have one or more boxes shaded to ensure that the programmers write in the correct positions.



I·C·T

RAPIDWRITE ENVIRONMENT DIVISION

PROG. NO. | SEQ. NO. | PROGRAM NAME | PROGRAMMER | DATE | SHEET NO.

E | I D

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

SOURCE – COMPUTER

PROG. NO. | SEQ. NO. | TYPE | COMPUTER NAME | I.A.S. SIZE | BACKING STORE SIZE

E | S O

OBJECT – COMPUTER

COMPUTER NAME | I.A.S. SIZE | BACKING STORE SIZE

E | O B

FILE - CONTROL ASSIGN | FILE NAME | TO | HARDWARE NAME | I = INPUT O = OUTPUT | SAME RECORD AREA AS FILE | NO. OF ALTERNATE AREAS | O = OPTIONAL

E | A S

- value 0 = 9 to identify an individual unit of its type
- C meaning Card
- M meaning Magnetic tape
- P meaning Paper tape
- W meaning Printer
- K meaning Key-board device

D U MP ON FILE ☐ AT EVERY END OF REEL OF FILE ☐

Printed in Great Britain

© International Computers and Tabulators Limited 1964

RAPIDWRITE DATA DIVISION

PROGRAM NAME | PROGRAMMER | DATE | SHEET NO.

1 2 3 4 5 | 6 7 8 9 10 11 | 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 | 69 70 71 72 73 74 | 75 76 77 78 79 80

PROG. NO. | SEQ. NO. | LEVEL NAME | BLOCK SIZE

D | F D

PROG. NO	SEQ. NO.	E	LEVEL	NAME	PICTURE	VALUE	REDEFINES	OCCURS

Appendix 4 1900 COBOL direct access verbs

The table below summarises the clauses and statements necessary for processing direct access files.

1 Verb	2 Opening Mode	3 How Data is Held on Device			4 Access Mode		5 Organization		6 Key		7 Verb must be Preceded by	8 May Intervene Between 7 and 1	9 Processing		Effect
		Serial	Sequen-tial	Ran-dom	Sequen-tial	Ran-dom	Index-ed	Direct	Actual	Sym-bolic			Auto-matic	Non Auto-matic	
DELETE	I/O	✓			✓						READ	WRITE			Deletes last record read
	I/O		✓		✓						READ	WRITE			
	I/O		✓			✓	✓				READ	WRITE	✓		
	I/O		✓			✓	✓			✓	SEEK			✓	Deletes record specified in SYMBOLIC KEY
	I/O			✓		✓		✓	✓	✓	Addr.Gen.Proc.	READ			Deletes record specified by SYMBOLIC & ACTUAL KEYS
READ	I,I/O	✓			✓										Reads next physical record
	I,I/O		✓		✓										Reads next record in key sequence order
	I,I/O		✓			✓	✓				*		✓		
	I,I/O		✓			✓	✓			✓	SEEK			✓	Reads record specified by SYMBOLIC KEY
	I,I/O			✓		✓		✓	✓	✓	Addr.Gen.Proc.				Reads record specified by SYMBOLIC & ACTUAL KEYS
RE-WRITE	I/O	✓			✓						READ	WRITE			Replaces last record read
	I/O		✓		✓						READ	WRITE			
	I/O		✓			✓	✓				READ	WRITE			
	I/O			✓		✓		✓			READ				
SEEK	I,I/O		✓			✓	✓			✓					Identifies bucket number using SYMBOLIC KEY. No effect
	I,I/O			✓		✓		✓							
WRITE	O	✓			✓										Writes specified record in next physical position
	I/O	✓			✓						READ	WRITE			Inserts specified record before last record read
	I/O		✓		✓						READ	WRITE			
	I/O		✓			✓	✓				READ	WRITE	✓		
	I/O		✓			✓	✓			✓	SEEK			✓	Inserts record in key sequence order (given by SYMBOLIC KEY)
	I/O			✓		✓		✓	✓	✓	Addr.Gen.Proc.				Inserts record in bucket indicated by ACTUAL KEY

Notes: 1 The processing of a verb applied to a file whose access mode is RANDOM and organization is INDEXED is non-automatic if it is preceded by a SEEK statement without an intervening READ: otherwise it is automatic.

2 A READ indicated by * can be preceded only by another READ or a REWRITE or an automatic DELETE.

Appendix 5 1900 Series COBOL example

Figure 1 - Data and Procedure Division Lines on 80-column Cards from Two Programs
Figure 2 - Data Maps
Figure 3 - Segment Listing and Program Maps
Figure 4 - Steering Lines for a CORRECT-COMPILE-CONSOLIDATE Run
Figure 5 - Compiler Listing of an XE13 compilation

Cobol program sheet title programmer sheet number date

Sequence No. 1 6 7 8 11 12 15 20 25 30 35 40 45 50 55 60 65 70 72 73 75 80 Identification

```
000100 FILE SECTION.
      110 FD  MASTER-FILE; BLOCK 6 RECORDS; LABEL RECORDS STANDARD; VALUE OF UPDATE-S
      120 ID "MASTER-STOCK"; ACTIVE-TIME IS 30; DATA RECORDS SUB-ASSEMB UPDATE-S
      130 -LIES. STANDARD-PARTS. UPDATE-S
      140  01 SUB-ASSEMBLIES. UPDATE-S
      150  02 CODE PICTURE X. UPDATE-S
      160  02 ASSEMBLY-SHOP-NO PICTURE 99. UPDATE-S
      170  02 ASSEMBLY-NAME PICTURE X(40). UPDATE-S
      180  02 QUANTITY. UPDATE-S
      190  03 IN-STORE PICTURE 9(4). UPDATE-S
      200  03 UNDER-TEST PICTURE 9(4). UPDATE-S
      210  03 UNDER-ASSEMBLY OCCURS 5 PICTURE 9(4). UPDATE-S
      220  02 FILLER PICTURE 9(9). UPDATE-S
      230  01 STANDARD-PARTS. UPDATE-S
                          (etc)

     3100 PROCEDURE DIVISION. SALES

     5400 CODE-CHECK. SALES
     5500 IF SALES-CODE EQUAL TO "B", ADD ITEM-1 TO TOTAL-SALES. SALES
     5600 IF SALES-CODE EQUAL TO "C" GO TO EXPORT-SALES. SALES
     5700 TARGET-CHECK. IF NEW-SALES LESS THAN TARGET DISPLAY "THE SA SALES
     5800 -"LES MANAGER MUST GO" PERFORM DISMISSAL GO TO RECESSION. SALES
     5900 IF NEW-SALES EQUAL TO TARGET DISPLAY "BULLS-EYE", TARGET BRANC SALES
     6000 -H-CODE GO TO NEW-TARGET-CALC. SALES
     6100 PERFORM GLAD-HAND. SALES
     6200 GO TO BONUS-CALC. SALES
```

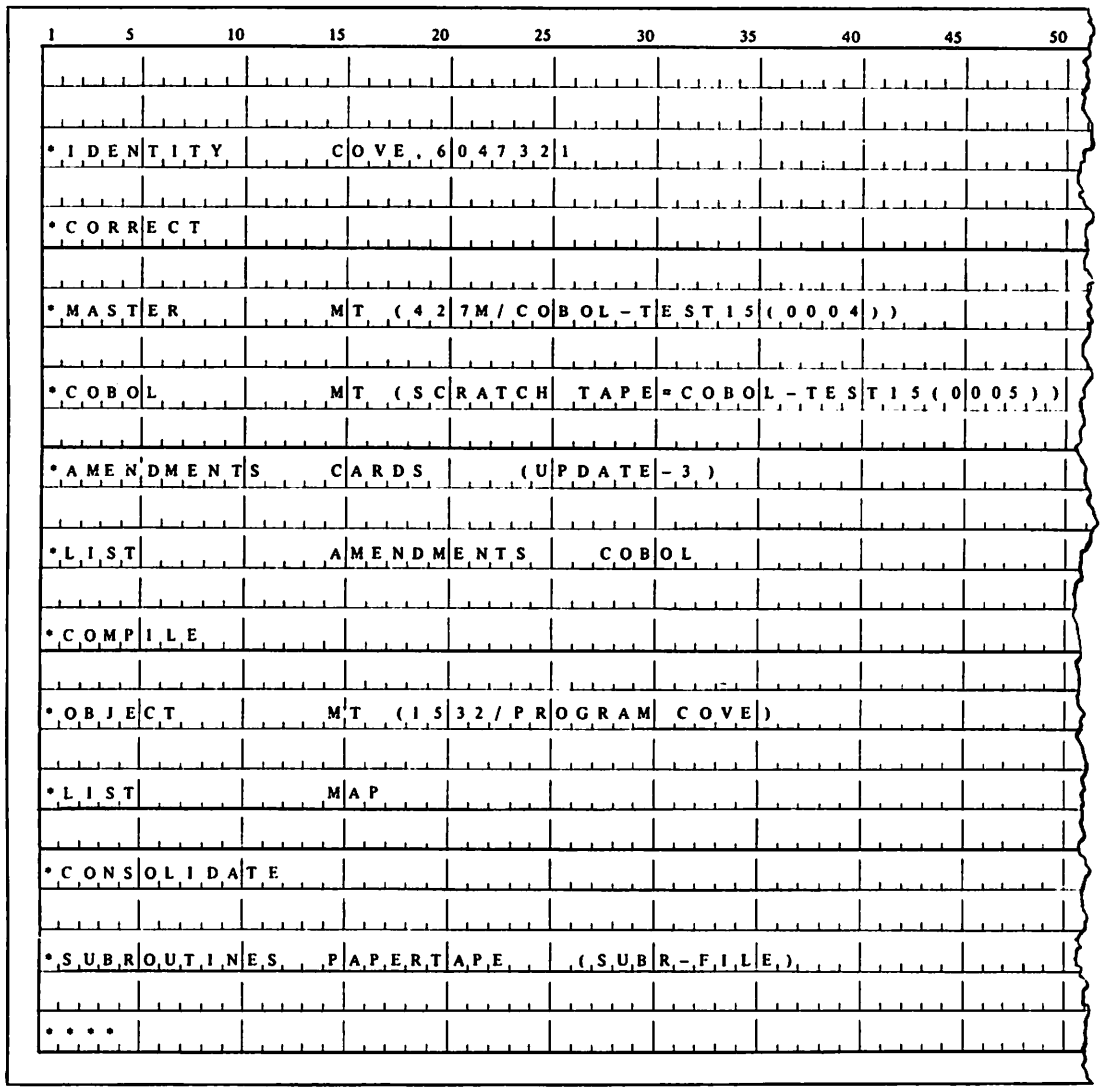
Figure 1 Data and procedure division lines on 80- column cards from two programs

```
47   51   54                                                    85   88
MAP FD MASTER-WORE                                             LV 2.1
MAP 01 CURRENT-STOCK                                           LV 21·0·0
MAP 02 AGENCY                                                  LV 21·0·0
MAP 03 BRANCH-CODE                                             LV 21·0·0
MAP 03 AREA-CODE                                               LV 22·1·0
MAP 03 ITEM-CODE                                               LV 23·2·0
MAP 02 QUANTITY                                                LV 24·0·0
MAP 03 WEIGHT                                                  LV 24·0·0
                (etc)
```

Figure 2 Data maps

```
0    4    9    14   19   24   29   34   39   44   49   54   59   64   69   74   79
MAP  PROGRAM NAME TST1
MAP  SECTION BRANCH-TEST-1                          OVERLAY 1.0
MAP      SEQ 004200      ADDRESS 00000.5
MAP      SEQ 004300      ADDRESS 00000.8
MAP      SEQ 004400      ADDRESS 00001.4
MAP  SECTION BRANCH-TEST-1                          OVERLAY 1.0   PROGRAM 1.47
MAP  SECTION BRANCH-TEST-2                          OVERLAY 1.1
MAP  SECTION MAIN-PATH                              PERMANENT
MAP  SECTION MAIN-PATH                              PERMANENT     PROGRAM 3.89
```

Figure 3 Segment listing and program maps



: The name of the source program is COVE, and the account number is 6047321.

: The source program is to be CORRECTed.

: The source program is on magnetic tape 427 labelled COBOL-TEST15, generation number 4.

: The new corrected source is to be output to a scratch tape 428, which is to be labelled as quoted.

: The amendment lines are on cards.

: A listing is required both of the amendment lines and the corrected source.

: The amended source is to be compiled.

: The object program is to be output on magnetic tape 1532 and already labelled PROGRAM COVE.

: A listing is required of the Data Maps, Program Maps and segments (if any).

: The object program is to be consolidated.

: Subroutines from a paper tape library named SUBRFILE are to be included in the consolidated object program.

: End Marker.

Figure 4 Steering lines for a CORRECT-COMPILE-CONSOLIDATE run

Figure 5 Compiler listing of an XE13 compilation

```
      DATE 18/10/67                              1900 COBOL COMPILER XE13 #556
*IDENTITY BACH
*COMPILE
*COBOL CARDS (BACH)
*OBJECT MT (SCRATCH TAPE=PROGRAM BACH)
*LIST COBOL MAP
*CONSOLIDATE
****
COMPILER SEQUENCING
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.
000300     BACH33.
000400 ENVIRONMENT DIVISION.
000500 CONFIGURATION SECTION.
000600 SOURCE-COMPUTER.
000700     ICT-1901.
000800 OBJECT-COMPUTER.
000900     ICT-1901.
001000     MEMORY 8000 WORDS.
001100 INPUT-OUTPUT SECTION.
001200 FILE-CONTROL.
001300 SELECT MASTER-FILE-A ASSIGN 1 TAPES.
001400 DATA DIVISION.
001500 FILE SECTION.
001600 FD  MASTER-FILE-A
001700     BLOCK 10 RECORDS
001800     LABEL RECORDS STANDARD
001900     VALUE OF ID IS "SDO PROJECTS" ACTIVE-TIME IS 40
002000     DATA RECORDS INFO.
002100 01  INFO-1.
002200 02  PARA-M  PIC 9 OCCURS 20.
002300 02  D-M     PIC X OCCURS 2.
002400 WORKING-STORAGE SECTION.
002500 77  C-COUNT-A  PIC 99  COMP SYNC  RIGHT.
002600 77  G1  PIC A9.
002700 77  G2  PIC X9.
002800 77  ABX PIC S1(47).
002900 77  ABY PIC S9V9(7).
003000 77  ABZ PIC S9(9)V999.
003100 77  ABXY PIC S9(9).
003200 01  INFO-2.
003300 02  SW-P  PIC 99 COMP.
003400 02  PS-PARA-M  PIC 9 OCCURS 20.
003500 02  PS-D-M     PIC X OCCURS 20.
003600 02  SW-J       PIC 9.
003700 02  SW-E       PIC 9 OCCURS 3.
003800 01  G-1   PIC 9,999  DISPLAY-1.
003900 01  INFO-3.
004000 02  G-2   PICTURE 9(4).
004100 01  INFO-4.
004200 02  ROB-1  PICTURE S9(6) DISPLAY-1.
004300 02  ROB-2  PIC     S1(47) COMP-1  SYNC RIGHT.
004400 77  WIMPEY  PIC 1(10).
004500 PROCEDURE DIVISION.
004600 BEGIN.
004700     IF SW-P IS NOT ZERO AND PARA-M (SW-P) IS ZERO, OR SW-J NOT
004800     ZERO AND D-M(SW-J) = SPACES, GO TO PARA-E.
004900     IF SW-E (C-COUNT-A) IS ZERO, GO TO PARA-E.
005000     IF SW-E (C-COUNT-A) IS ZERO  GO TO PARA-E.
```

```
005100      IF SW-E (C-COUNT-A) ZERO GO TO PARA-E.
005200      IF SW-P IS NOT ZERO AND PARA-M (SW-P) IS ZERO OR SW-J NOT
005300      ZERO AND D-M(SW-J) = SPACES GO TO  PARA-E.
005400      IF SW-P IS NOT ZERO AND PS-PARA-M (SW-P) IS ZERO OR SW-J NOT
005500      ZERO AND PS-D-M (SW-J) = SPACES GO TO PARA-E.
005600 PARA-E.
005700      MOVE 2 TO G-2.
005800 PARA-G.
005900      PERFORM PARA-TEST.
006000      MOVE 10 TO G-2.
006100      PERFORM PARA-TEST.
006200      MOVE 180 TO G-2.
006300      PERFORM PARA-TEST.
006400      MOVE 999 TO G-2.
006500      PERFORM PARA-TEST.
006600      MOVE 1000 TO G-2.
006700      PERFORM PARA-TEST.
006800 PARA-R.
006900      SUBTRACT ROB-1 FROM ROB-2.
007000      STOP RUN.
007100 PARA-W.
007200      MOVE ROB-2 TO  G-2(WIMPEY).
007300 PARA-TEST.
007400      MOVE G-2 TO G-1.
007500      DISPLAY G-2, " = ", G-1.
007600      MULTIPLY ABX BY ABY GIVING ABZ ROUNDED.
007700      MOVE ABZ TO ABXY.
****
ERROR   SEQ 004700  REF 012 INVALID SYNTAX  0232
ERROR   SEQ 004900  REF 012 INVALID SYNTAX  0606
ERROR   SEQ 005000  REF 012 INVALID SYNTAX  0606
ERROR   SEQ 005100  REF 012 INVALID SYNTAX  0606
ERROR   SEQ 005200  REF 012 INVALID SYNTAX  0232
ERROR   SEQ 005400  REF 012 INVALID SYNTAX  0232
ERROR   SEQ 007200  REF 003 UNDEFINED NAME - MOVE
ERROR   SEQ 007200  REF 047 TO ALONE
                                    MAP FD MASTER-FILE-A
                                    MAP 01 INFO-1                          0,0,0
                                    MAP 02 PARA-M                          0,0,0
                                    MAP 02 D-M                             5,0,0
                                    MAP 77 C-COUNT-A              LP      46,2,5
                                    MAP 77 G1                     LP      47,0,0
                                    MAP 77 G2                     LP      48,0,0
                                    MAP 77 ABX                    LP      49,0,0
                                    MAP 77 ABY                    LP      51,0,0
                                    MAP 77 ABZ                    LP      53,0,0
                                    MAP 77 ABXY                   LP      56,0,0
CAUTION SEQ 003400  REF 736 NEEDS FILLER (A)
                                    MAP 01 INFO-2                 LP      59,0,0
                                    MAP 02 SW-P                   LP      59,0,0
                                    MAP 02 PS-PARA-M              LP      59,2,0
                                    MAP 02 PS-D-M                 LP      64,2,0
                                    MAP 02 SW-J                   LP      69,2,0
                                    MAP 02 SW-E                   LP      69,3,0
                                    MAP 01 G-1                    LP      71,0,0
                                    MAP 01 INFO-3                 LP      73,0,0
                                    MAP 02 G-2                    LP      73,0,0
CAUTION SEQ 004300  REF 736 NEEDS FILLER (A)
                                    MAP 01 INFO-4                 LP      74,0,0
```

```
                                        MAP 02 ROB-1                    LP    74.0.0
                                        MAP 02 ROB-2                    LP    76.0.0
                                        MAP 77 WIMPEY                   LP    78.0.0
                                  FILE-LIST   MASTER-FILE-A             LV    43
CAUTION SEQ 006900  REF 408 COMPOSITE TOO BIG
CAUTION SEQ 007600  REF 408 COMPOSITE TOO BIG

MAP  PROGRAM  NAME  BACH

MAP SECTION COBOL                          PERMANENT
MAP      SEQ 004600      ADDRESS 000002
MAP      SEQ 004700      ADDRESS 000002
MAP      SEQ 005200      ADDRESS 000003
MAP      SEQ 005400      ADDRESS 000004
MAP      SEQ 005600      ADDRESS 000005
MAP      SEQ 005700      ADDRESS 000006
MAP      SEQ 005800      ADDRESS 000007
MAP      SEQ 005900      ADDRESS 000007
MAP      SEQ 006000      ADDRESS 000011
MAP      SEQ 006100      ADDRESS 000013
MAP      SEQ 006200      ADDRESS 000017
MAP      SEQ 006300      ADDRESS 000019
MAP      SEQ 006400      ADDRESS 000023
MAP      SEQ 006500      ADDRESS 000025
MAP      SEQ 006600      ADDRESS 000029
MAP      SEQ 006700      ADDRESS 000031
MAP      SEQ 006800      ADDRESS 000035
MAP      SEQ 006900      ADDRESS 000035
MAP      SEQ 007000      ADDRESS 000078
MAP      SEQ 007100      ADDRESS 000082
MAP      SEQ 007300      ADDRESS 000082
MAP      SEQ 007400      ADDRESS 000082
MAP      SEQ 007500      ADDRESS 000116
MAP      SEQ 007600      ADDRESS 000133
MAP      SEQ 007700      ADDRESS 000209
MAP SECTION COBOL                          PERMANENT    PROGRAM    223
STATISTICS: *DATA      =   24 X  5.0
STATISTICS: *PROCEDURE=    6 X  6.3
STATISTICS: *LOWER=  68 WORDS
CORE   1024
       LV 45          LP 94   LT 196          UP 439          UV 480
MAP  PERMANENT PROGRAM BEGINS AT   197
41         420  COBO
CAUTION SEQ 007700  REF 619 TOTAL OF    8 ERROR(S) SIGNALLED
```

Appendix 6 General description of the compilers

COMPILER PROGRAM UNITS - NAMES AND ACTION

The compiler program is divided into two main parts corresponding to two phases of compilation: *Analysis* and *Generation*. There are also three other phases which may or may not be called for by the user according to the needs of a particular compilation:

CONSOLIDATE, which combines the semi-compiled object program segments, and library routines if required, into a single, loadable object program.

CORRECTION, which is used to make corrections to the COBOL source program and/or to preserve the corrected source on magnetic tape.

RAPIDWRITE-TO-COBOL TRANSLATOR, which forms part of the Analysis phase when the input is in Rapidwrite. All phases of the compiler may flag errors in the course of processing the source program.

The analysis phase

The Analysis phase validates the COBOL program word by word and statement by statement, encodes the Environment and Data Division into tables, and translates the Procedure Division into a form – 'Simplexes' or simple expressions – from which the Generation phase can produce object code.

The constituents of the Analysis phase are:

READ-A-WORD, which

1 isolates and validates each COBOL word or punctuation mark in turn,

2 using a table of COBOL key words, identifies it as either a key word, a literal, a user-defined name or an unidentified word (such as a Picture),

3 outputs an identifying code, the name itself, and the number of characters in it.

Depending on which version of the compiler is being used, this information is either passed on in store, or output to magnetic tape.

ANALYZER, which, using the identities supplied by Read-a-Word, examines each COBOL statement in turn and compares it with the entries in a Syntax Table to see if it is valid. If it is valid, the statement is transformed into a 'tree structure'; if it is not valid, an error message is printed and the statement is converted to a valid form where possible or otherwise is ignored. At the same time, a Name table of all the entries in the Data Division is composed, and when the Procedure Division is processed, a check is made that all data names referenced in procedure statements have in fact been defined in the Data Division.

PROPERTIES GENERATE, which

1 accepts as input the 'trees' produced by Analyzer for the Data and Environment Divisions,

2 creates from them a Properties table holding in coded form all the information about each field (size, usage, editing, etc.), and a Block table holding information about buffer areas for magnetic tape files,

3 assigns object program addresses to all fields while checking that they are validly and consistently described.

SIMPLEX GENERATE, which accepts as input from Analyzer the statements of the Procedure Division in 'tree' form, and outputs them in the form of Simplexes. Further validation checks are performed, and a Paragraph table and Segment table are produced. Data names referred to in COBOL statements are replaced by the addresses of corresponding entries in the Properties table. Thus, by the end of Simplex Generate, the elements of the COBOL source program are completely identified and recoded in numeric form. The Paragraph and Segment tables are carried forward in store; the Properties table is held in store or on magnetic tape, depending on the size of store available to the compiler.

APPEND At the end of Simplex Generate, the compiler decides whether or not to eliminate the Properties table by 'Appending' to each Simplex the properties of all data fields referred to in the Simplex. The decision will depend on the size of store available *at the moment*, i.e. the initial size of store less the amount used by the tables that have been produced (which are themselves a function of the size and complexity of the COBOL source program). The compiler has to take into account that if a large Properties table is held in store (i.e. no 'Appending' takes place) the Generation phase may consequently have to be split up into more passes than would otherwise be the case and balance the time so lost against the time gained by not having a separate Append pass.

This decision about whether to split the Generation phase into more than one pass, together with a decision taken before the Analysis phase (according to the availability of store) on how to combine the various Analysis units for the compilation, enables the compiler to optimize itself according to the particular circumstances of the run. This is especially useful on multiprogramming machines if the user wishes to vary the amount of core store available to the compiler on different occasions.

The generation phase

The Generation phase comprises a control routine, a table of generator addresses, and a set of generators corresponding to the Simplexes, together with standard subroutines and 'sub-generators' (e.g. for producing object code to do specific tasks such as editing, aligning arithmetic fields, allocating working-storage). The Generation phase accepts as input from Analysis phase:

the Paragraph table (in store, or if Append has taken place, may be on magnetic tape),

the Segment table (in store),

the Block table (in store),

the Properties table, if Append has not taken place (in store),

the Simplexes, with or without Appended properties (on magnetic tape).

A switch will have been set to indicate whether or not Appending has occurred. If the compiler selects a Generate phase of more than one pass, each pass will contain the same control routine but a different set of generators and sub-generators and a table of generator addresses corresponding to them. Each pass will

1 process, i.e. generate semi-compiled code, for all Simplexes for which it has generators,

2 pass on all other Simplexes unprocessed, on the same tape as the code produced by 1,

3 on meeting an END Simplex, which marks the end of the source program, load the generators and table for the next pass.

Great stress has been laid on the production of optimal code. Rather than produce keys and code to prime generalized run-time subroutines, the generators attempt to produce optimal code for each and every particular situation. This rarely takes more space and is much faster than the subroutine approach.

It will be realized that whether a one-pass or a multi-pass Generation phase is used, certain types of Simplex can only be processed in the last pass. For example, run-time addresses can only be assigned to paragraph names when all the code for the rest of the source program has been generated, that is, when all other addresses have been calculated. In addition, the last pass prints one or more Terminal messages (see page 239) and performs a number of other end-of-run functions.

The Rapidwrite-to-COBOL translator

If the source program is in Rapidwrite, it is translated into COBOL and the output passed on to Read-a-Word. The latter identifies the COBOL words, and thereafter compilation proceeds as for COBOL input. The Translator may also be used independently of the rest of the compiler to produce a COBOL version of a Rapidwrite source. There will be a printer listing of the Rapidwrite input and/or the COBOL version only if the appropriate Steering lines are used (see page 290).

The consolidate phase

The output from the Generation phase is an object program in semi-compiled form on magnetic tape, paper tape or cards. If the user wishes to consolidate all the program segments so produced into a single, loadable program, he instructs the compiler to do this by means of a *CONSOLIDATE Steering line (see page 190). The compiler then uses its own built-in Consolidator to produce an object program on magnetic tape, paper tape or cards whose order is:

Program

Loader (which may be an overlay loader)

Consolidated cue list

Note: An overlay object program can only be on magnetic tape.

At the same time, the Consolidator will incorporate in the object program:

1 any routines that may be required from the Magnetic Tape Housekeeping package,

2 the standard overlay package, if the source program was divided into overlay segments,

3 any Own Code routines (user's routines or library subroutines) that may have been called for by ENTER verbs in the COBOL source (see Chapter 17),

4 any standard compiler subroutines that may be required.

The resultant consolidated program is then ready to be loaded and run. If the *LOAD Steering line has not been used there will be a halt, and the message

CONSOLIDATED# ppppnnnnnnnn

(where pppp is the program name and n..n is the account number) will be displayed on the console typewriter. Peripherals should be loaded/unloaded by the operator in accordance with user's instructions, and the object program is entered as described on page 247.

The correction phase

The Correction phase is entered only if the *CORRECT Steering line is used. It performs two main functions:

1 It makes insertions, deletions and corrections to the source program according to the Correction Lines supplied as parameters (on cards or paper tape).

2 It transcribes the source program onto magnetic tape, paper tape or cards.

Full details are given on page 262.

Compilation times

Compilation times will vary according to a number of factors, including:

1 Which version of the compiler is used,

2 The core store made available to it,

3 The input and output media (cards, paper tape or magnetic tape),

4 The speed of the tape decks,

5 What printed listings are required,

6 The complexity and size of the source program,

7 The 'mode' of compilation, e.g. whether Corrections are being applied.

IMPLICATIONS OF COMPILING ON LARGER MACHINES AND CONFIGURATIONS

Three main advantages may be obtained by compiling on machines with a larger store and/or more magnetic tape decks than required for a minimum configuration.

1 The larger the store, the faster compilation will be, because there will be fewer passes.

2 Similarly, the compiler-generated tables can be larger, which means there can be more data and procedure names in the COBOL source.

3 If extra tape decks are available over and above the minimum four, there will be a considerable saving in mount/dismount times, e.g..when preserving the COBOL source or corrected source, mounting a tape containing user's subroutines etc. (see page 190).

Index

ACCEPT 72-73
ACCEPT FILE KEYS 72,73,143-144
ACCESS MODE 26,28-29
ACCESS MODE IS RANDOM 26,29,30
ACCESS MODE IS SEQUENTIAL 26,28-29,30
ACTIVE-TIME 36,38
ACTUAL KEY 26,29,30
ADD 74,114
ADVANCING 103,110,112
AFTER 103
ALPHABETIC 44,69,117,119,123
ALPHANUMERIC 44,117,119,122-123
ALTER 75
AN 44
Analysis Phase 289
Analyzer 289
AND 70
Append 290
APPLY .... TO INDEX-BUFFER ON 31-32
APPLY....TO TRANSFER-REPLY ON 31-32
ASSIGN OBJECT-PROGRAM 23
AT END 93,97
AUTHOR 19

BEFORE 103
Binary Computations 112
BLANK WHEN ZERO 45,126
BLOCK CONTAINS 36,37,131

CALL 76
CARD-PUNCH 28
CARD-READER 23,27
CHANNEL-n 24,103
Character Peripheral Devices 127
Character Set
  Data Character Set 271
  Language Character Set 14
Check List 184-185
CHECK PROTECT 45
CHEQUE PROTECT 45
Cheque Protection 126
CLASS 44
CLOSE 77,144
CODASYL xv
Compilation Times 291
Compilers
  XE11 1,4,5-11
  XE13 2,4,5-11
  XE15 1,4,5-11
  XE17 2,4,5-11
  XE18 1,4,5-11
  XE19 2,4,5-11
  XE20 3,4,5-11
Composite 109
COMP 61
COMP-1 61
COMPUTATIONAL 59,60-61,118,120,133
COMPUTATIONAL-1 60-61,118-119,120-121
Condition Names 15,63
Conditions 67-71
  Class conditions 69-70
  Compound conditions 70-71
  Condition-name conditions 70
  Relation conditions 67-69
  Sign conditions 69
  Switch-status conditions 70
CONFIGURATION SECTION 21-25
Consolidate Phase 290-291
CONSTANT SECTION 33,40
Continuation Indicator 18
COPY 22,23,24,26,31,36,42,78,107
CORE-EDS 23
CORE-MT 23
Correcting the Source Program 262-263
  Deletions 262-3
  Insertions 263
  Replacements 263-4
Correction Phase 291
Currency Symbol 124

DATE COMPILED 19
Data Description Entry 34
DATA DIVISION 33
Data Division 33-64
Data Names 15,66,131
DATA RECORDS 36,38
Data References 66
DATE WRITTEN 19
Decimal Computations 110
DELETE 28,29,30,79
Deletions 263
DEPENDING ON 88
Direct Access Devices 129-130
Direct Access Verbs 279
DISPLAY 80-81
DISPLAY-n 40-59,117-118,119-120
Distribution of Input Data 135-136
DIVIDE 82,110-111,113,114-115
Dumping 246-247

Editing 122-126
Editing Clauses 45
EDS 23,27
EDS-* 31
Elementary field 34
Elementary fields 117-121

ELSE 89
ENTER 83-85,144-145
ENVIRONMENT DIVISION 21
Environment Division 21-32
EQUAL(S) 69
Error Messages 199
  Format and Parameter Errors 201-204
  Language Errors 204-233
  Run time error messages and halts 247-250
EXAMINE 86
EXCEEDS 68
EXIT 87

FD 36
FDS 27
FDS-* 31
FD Skeleton 36
Field 34
Figurative Constants 16,66,122
File 34
FILE CONTAINS 36,37
FILE-CONTROL 26-30
File Description 36-38
File Names 36
File Numbering 247
FILE SECTION 33,35-38
FILLER 46,133,134
FLOAT CURRENCY SIGN 45
Format Disciplines 131
Fullstop 14,139

GENERATION-NO 37
Generation Phase 290-291
GO TO 88
GREATER 69
Group field 34
Group fields 122

Hardware Name 24,26

Identification Columns 18
IDENTIFICATION DIVISION 19
Identification Division 19
IF 89
INPUT-OUTPUT SECTION 26-32
I-O-CONTROL 31-32
Insertion Characters 123
Insertions 263
INSTALLATION 19
INVALID KEY 79,97,98,103,104

JUSTIFIED 47

Keywords and Symbols 16

Labels 246
LABEL RECORD 36,37
Language Character Set 14
LEAVING integer PLACES 45
LESS 69
Level-number 42
Limits, Compiler 265-268
LINES 103
LINKAGE SECTION 33,41
Listings
  Consolidator 199
  Data Maps 197
  Initial Messages 197
  Program Maps 197
  Segment 197
  Source Program 197
Literals 66,121-122,143
  Non-numeric 15,66
  Numeric 15,66
  Figurative Constants 16,66,122

Magnetic tape devices 128-129
MCF 27
MCF-* 31
MEMORY SIZE 23,105
Messages
  Compiler 193
  Error 199-237
  General Purpose Loader 193
  Other Compiler Time 238-239
  Overlay Packages 194
  Run-time error messages and halts 247-250
  Terminal 239
Mixed Computations 113-115
Mnemonic-name 24
Monitoring the Source Program 261-2
MOVE 90,113,117-126
MULTIPLE REEL 26,28
MULTIPLY 91,110,112,114

NEGATIVE 69
NEXT SENTENCE 89
NOT 70
Notation 18
NOTE 92
NUMERIC 44,69,119

OBJECT-COMPUTER 23
Object Program 241-254
  Abortive Object Runs 251
  Dumping and Restarts 246
  File Numbering 247
  Format 241
  Labels and Sentinels 246
  *LOAD Steering Line 251
  *LOWER Steering Line 242-243
  Operating Instructions 247
  Peripheral Handling 243-246
  Run-time error messages and halts 247-250
  Run-time subroutines 251-254
  Storage, allocation of 241-242
OCCURS 48
OFF STATUS 24
ON STATUS 24
OPEN 93
Operating Instructions 247
Optional words 16
OR 70
ORGANIZATION 26,29-30
ORGANIZATION IS DIRECT 26,29,30
ORGANIZATION IS INDEXED 26,29,30

PAPER-READER 23,28
PAPER-PUNCH 28
Paper Tape 127
  End Marker 132
  Inter-block type 132
  Short Data Blocks 133
Paragraph 65
Parenthesis
  Left 14
  Right 14
PERFORM 94-96,145-147
Peripheral Handling in
Object Program 244-246
Philosophy of COBOL xvi
PICTURE 49-52
POINT LOCATION 53
POSITIVE 69
PRINTER 26
Printing 128
Procedure Division 65-104
PROCEDURE DIVISION 65
Procedure Names 15
Procedures 65
PROCESSING MODE IS SEQUENTIAL 26,29
PROGRAM-ID 19
Program name 19
Program sheet 17
Program Structure 13
Program Text 18
Properties Generate 289
Punched Cards 127
Punctuation 14

Qualified data names 17,66

Rapidwrite 169-179,275-277
Rapidwrite-to-COBOL-Translator 289
READ 28,29,30,97-98
Read-a-word 289
Record 33
RECORD CONTAINS 36
Record Count Word 132
Record Description 42-62
RECORDING MODE 36
REDEFINES 54,136-139
REMARKS 19
RENAMES 55
Replacements 263-4
REPLACING 22,23,24,26,31,42,78,86,107
Report Signs 124-125
Representation of Data on
External Media 127-130
RERUN 31-32
RESERVE 26,28
Reserved Words 273-274
Restarts 246
REWRITE 28,29,30,99
ROUNDED 74,82,91,102
Rounding 109

SAME 31,32
Section 65,105-106
SECTION 13,65,105
SECURITY 19
SEEK 29,30,100
Segment 105-106
Segmentation 105-106,142,183
SEGMENT-LIMIT 23
SELECT.....ASSIGN 26-28
Semi-colon 14
Sentence 65
Sentinels 246
SEQUENCED ON 36,38
Separators 65
Sequence Number 17
SIGNED 56
Simplex Generate 289
SIZE 57
Size Error 109
SIZE ERROR 74,82,91,102
Sorting 153-160
SOURCE-COMPUTER 22
Source Program
  Compiler Operating Instructions 191-194
  Format 181-182
  Monitoring 261-262
  Order of Input 191
  Preparation 184-185
SPECIAL-NAMES 24
Statement 66
  Compiler Directing 66
  Conditional 66
  Formats 71-104
  Imperative 66
Steering Lines 185-191
  *LOAD 251
  *LOWER 242-243
Sterling Computations 111-112
STOP 101,147
Storage
  Allocation of 63-64
  In object program 241-243
Subroutines
  CCLOSE 153
  CDELETE 153
  CEXTEND 153
  COBDIST 160-167
  COBDISTM 160-164
  COBOL Subroutines 255
  COBPAK 160-167
  COBPAKM 160-167
  COBOLSORT 153-160
  COPENEND 151
  COPENINOUT 151
  COPENINPUT 150
  COPENOUTPUT 150-151
  CRDELETE 153
  CREAD 151
  CREPLACE 152
  CRESET 153

CRETURN 153
CRINSERT 152
CRREAD 152
CSDELETE 153
CSDELETEA 153
CSINSERT 152
CSINSERTA 152
CSREAD 151
CSREADA 151
CWRITE 152
"Force-in" bit 258
Format 258
Order of Subroutines 258
Pause Block 259
PUNCHN 166
Run time Subroutines 251-254
Subscripted data names 66-67
SUBTRACT 102,110,112,114
SYMBOLIC KEY 26,29,30
SYNC 58
Synchronization 64,134-135
SYNCHRONIZED 55,121

TALLY 86
TALLYING 86
TAPE-* 31
TAPES 23-26
Testing 261-262
TIMES 94,95
TYPE 24,26,27-28
Type-numbers 27

UNTIL 94,95
USAGE 59-61
USING 65,83

Validation of Data 136
VALUE 62
VALUE OF ID 36,38
Variable length fields 127-129,160-167
VARYING 94,95-96

WORKING-STORAGE section 33,39
WRITE 29,30,103-104,147
ZERO 69
ZERO SUPPRESS 45
Zero Suppression 125-126

ICL

1900 Series

COBOL

First Edition March 1968

Amendment list 6
incorporating User Notices numbers 2 and 35

Each amendment list contains one or more numbered instructions to replace one or more existing pages or to add one or more new pages.

When a page is amended, significant technical changes on the re-issued page will be indicated by a vertical line in the margin against the changed passages. Any lines on the re-issued page which remain from a previous amendment, will be removed. New chapters or completely revised chapters will not be marked with amendment lines.

The date of issue appears at the foot of all new pages and re-issued pages in the form (month, year).

1	Contents	Remove and destroy pages v to viii. Insert new pages v to viii.
2	Chapter 4	Remove and destroy pages 29 and 30. Insert new pages 29 and 30.
3	Chapter 5	Remove and destroy pages 41 and 42. Insert new pages 41 and 42.
4	Chapter 6	Remove and destroy pages 77 and 78. Insert new pages 77 to 78.1.
5	Chapter 6	Remove and destroy pages 93 and 94. Insert new pages 93 to 94.1.
6	Chapter 11	Remove and destroy pages 127 to 130.1. Insert new pages 127 to 130.1.
7	Chapter 13	Remove and destroy pages 154.1 and 154.2. Insert new pages 154.1 and 154.2.
8	Chapter 15	Remove and destroy pages 185 and 186. Insert new pages 185 and 186.
9	Chapter 17	Remove and destroy pages 247 to 250. Insert new pages 247 to 250.
10		Remove and destroy User Notices 2 and 35.
11		Update the amendment record and file this list at the back of the manual.

©International Computers Limited
ICL House, Putney, London, S.W.15

4082(8.70)

ICL

1900 Series

COBOL

First Edition March 1968

Amendment list 5
incorporating User Notices numbers 16, 19, 20 and 27.

Each amendment list contains one or more numbered instructions to replace one or more existing pages or to add one or more new pages.

When a page is amended, significant technical changes on the re-issued page will be indicated by a vertical line in the margin against the changed passages. Any lines on the re-issued page which remain from a previous amendment, will be removed. New chapters or completely revised chapters will not be marked with amendment lines.

The date of issue appears at the foot of all new pages and re-issued pages in the form (month, year).

1	Chapter 1	Remove and destroy pages 5 to 8. Insert new pages 5 to 8.1.
2	Chapter 4	Remove and destroy pages 21 to 24. Insert new pages 21 to 24.
3	FD SKELETON	Remove and destroy pages 37 and 38. Insert new pages 37 and 38.
4	JUSTIFIED PICTURE REDEFINES	Remove and destroy pages 47 to 54. Insert new pages 47 to 54.
5	SIZE	Remove and destroy pages 57 and 58. Insert new pages 57 and 58.
6	Chapter 6	Remove and destroy pages 71 and 72. Insert new pages 71 and 72.
7	MULTIPLY	Remove and destroy pages 91 and 92. Insert new pages 91 and 92.
8	Chapter 8	Remove and destroy pages 107 to 108.4. Insert new pages 107 to 108.4.
9	Chapter 9	Remove and destroy pages 111 and 112. Insert new pages 111 and 112.
10	Chapter 11	Remove and destroy pages 127 to 130. Insert new pages 127 to 130.
11	Chapter 12	Remove and destroy pages 133 and 134. Insert new pages 133 and 134.
12	Chapter 12	Remove and destroy pages 137 and 138. Insert new pages 137 and 138.
13	Chapters 12 and 13	Remove and destroy pages 143 to 150. Insert new pages 143 to 150.
14	Chapter 13	Remove and destroy pages 153 and 154. Insert new pages 153 and 154.
15	Chapter 13	Remove and destroy pages 155 and 156. Insert new pages 155 and 156.
16	Chapter 15	Remove and destroy pages 181 to 188. Insert new pages 181 to 188.
17	Chapter 16	Remove and destroy pages 203 to 204.2. Insert new pages 203 to 204.2.
18	Chapter 16	Remove and destroy pages 207 to 210. Insert new pages 207 to 210.
19	Chapter 16	Remove and destroy pages 215 to 216.1. Insert new pages 215 to 216.1.
20	Chapter 16	Remove and destroy pages 219 and 220. Insert new pages 219 to 220.1.
21	Chapter 16	Remove and destroy pages 229 to 237.1. Insert new pages 229 to 237.1.
22	Chapter 16	Remove and destroy pages 237.6 to 240. Insert new pages 237.6 to 240.
23	Chapter 17	Remove and destroy pages 247 and 248. Insert new pages 247 and 248.
24	Chapter 17	Remove and destroy pages 250.3 and 250.4. Insert new pages 250.3 and 250.4.
25	Chapters 19 and 20	Remove and destroy pages 263 to 268. Insert new pages 263 to 268.
26	Appendix 2	Remove and destroy pages 273 and 274. Insert new pages 273 and 274.
27		Remove and destroy User Notices 16, 19, 20 and 27.
28		Update the amendment record and file this list at the back of the manual.

©International Computers Limited
ICL House, Putney, London, S.W.15

4082(6.70)

1900 SERIES

ICL

COBOL

First Edition March 1968

Amendment List 4
incorporating User Notices COBOL 17, 21 and 31.

1 Remove and destroy pages v to xiii. Replace by new pages v to xiii.

2 Remove and destroy pages 1 to 12. Replace by new pages 1 to 12.

3 Remove and destroy pages 23 to 32. Replace by new pages 23 to 32.2.

4 Remove and destroy pages 35 to 38. Replace by new pages 35 to 38.1.

5 Remove and destroy pages 41 and 42. Replace by new pages 41 and 42.

6 Remove and destroy pages 47 and 48. Replace by new pages 47 and 48.

7 Remove and destroy pages 65 to 76. Replace by new pages 65 to 76.2.

8 Remove and destroy pages 89 to 92. Replace by new pages 89 to 92.

9 Insert new pages 98.1 and 98.2 after page 98.

10 Insert new pages 100.1 and 100.2 after page 100.

11 Remove and destroy pages 101 and 102. Replace by new pages 101 and 102.

12 Remove and destroy pages 107 and 108. Replace by new pages 107 to 108.4.

13 Remove and destroy pages 127 to 132. Replace by new pages 127 to 132.

14 Remove and destroy pages 139 and 140. Replace by new pages 139 to 140.1.

15 Remove and destroy pages 143 to 154. Replace by new pages 143 to 154.2.

16 Remove and destroy pages 159 and 160. Replace by new pages 159 to 160.7.

17 Remove and destroy pages 167 to 168.4. Replace by new page 167.

18 Remove and destroy pages 189 and 190. Replace by new pages 189 to 190.1.

19 Remove and destroy page 204.1. Replace by new pages 204.1 and 204.2.

20 Remove and destroy pages 211 to 216. Replace by new pages 211 to 216.1.

21 Remove and destroy pages 221 to 226. Replace by new pages 221 to 226.1.

22 Remove and destroy pages 229 and 230. Replace by new pages 229 to 230.1.

23 Remove and destroy pages 233 to 234.2. Replace by new pages 233 to 234.2.

24 Remove and destroy pages 237.2 and 237.3. Replace by new pages 237.2 and 237.3.

25 Remove and destroy pages 249 to 252. Replace by new pages 249 to 252.

26 Insert new page 254.1 after page 254.

27 Remove and destroy pages 273 and 274. Replace by new pages 273 and 274.

28 Remove and destroy pages 292.1 to 292.9.

29 Remove and destroy User Notices 17, 21 and 31.

International Computers Limited
ICL House, Putney, London, S.W.15

4082(3.70)

AMENDMENT RECORD

List Number *Item Numbers*

4	1 *to* 29
5	1 *to* 28
6	1 *to* 11
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to

Should any additions or amendments to this Manual be necessary, they will be issued in the form of Amendment Lists. When such changes have been incorporated, an appropriate entry should be made in the table on the left and the pink instruction sheet(s) should be filed behind this page.

ICL International Computers Limited

User notice

28/4/71

MANUAL (NOTICE NO.)

4082 COBOL (61)

File one copy of this notice with each of the manuals indicated.

CONTENTS

	Page
USER NOTICES	1
CACTION/3	2
COBOL SUBROUTINES	2
ERROR MESSAGES	7

USER NOTICES

Obsolete user notices

With the publication of this user notice, the following user notices to *COBOL* become obsolete and should be deleted:

10, 11, 22, 23, 25, 29, 30, 32, 34, 36, 40, 42, 43, 45, 46, 48, 49, 53, 54 and 56.

User notices about compilers

#XEKA/B

With the publication of User Notices (1) and (2) to *COBOL Compilers* (Edition 1, TP4236), all the information on compiling with #XEKA and #XEKB has been transferred to *COBOL Compilers*. This makes the following parts of user notices redundant:

51(last item, 52 (from page 12, paragraph -1, 'The sort will search...' to page 14, item E, '... DISPLAY: NNNN RECORDS SORTED') and 57 (page 2, paragraph 2, '7 The duplication of...', to the end of the page)

OTHER COMPILERS

Information on the other compilers is available in the following user notices:

#XEA3	37
#XE20	39 and 41 (first two items)
#XE1B	50

Details of the subroutines on the M.L.T. applicable to each of these compilers is given in User Notice *COBOL* (58).

Current user notices

The following user notices and parts of user notices to *COBOL* are still current:

37, 39, 41 (first two items), 47, 50, 51 (first three items) 52 (from the start to page 13, paragraph -2 and from page 14, paragrpah -4 to the end), 55, 57 (from the start to page 2, paragraph 1 and from page 3 to the end), 58, 59, 60 and 61.

CACTION/3

This version of CACTION does not zeroize the record length if an exception condition has arisen.

COBOL SUBROUTINES

This section is intended to supplement the information on subroutine facilities given in *COBOL* (see *Identification Division*, page 19, *Linkage Section*, page 41, *Procedure Division*, page 65, *CALL*, page 76 and *EXIT*, page 87).

In the examples given COBOL statements not relevant to the example have generally been omitted, for example, IDENTIFICATION DIVISION or ENVIRONMENT DIVISION. However, when actually writing a subroutine, all statements not marked as optional in *COBOL* must be included.

Language

SPECIAL COBOL OPTIONS FOR SUBROUTINE COMPILATION

The CALL verb

As explained above, it is necessary to pass the addresses of fields used by two distinct segments as parameters from one to the other. This is the function of the CALL verb. For example, the statement:

CALL SUBR USING parameter-1... parameter-*n*

causes an entry to be made to the subroutine SUBR and the addresses of parameters 1 to *n* to be passed across to it.

The way such statements are translated into machine code is described in Chapter 18 of *COBOL*.

The Linkage Section

The Linkage Section is necessary because the calling segment and the called subroutine are compiled independently; thus no information about the fields defined in one segment is available to the other at the time of compilation.

Suppose, for example, that the calling segment contains the statement:

```
CALL SUBR USING A, B
```

where A has the description:

```
PIC X (10)
```

B has the description:

```
PIC X(10)
```

and the called subroutine, SUBR, contains the statement:

```
MOVE A TO B.
```

The subroutine needs to know the descriptions of A and B in order to decide what sort of MOVE to generate. Since it has no access to the description of these fields in the calling segment, the description must be repeated in the called subroutine, in the Linkage Section.

The Linkage Section of the subroutine SUBR will look like this:

```
LINKAGE SECTION.
      77 A PIC X(10)
      77 B PIC X(10)
PROCEDURE DIVISION.
```

Notice that the names of fields in the called subroutine can be the same as the names in the calling segment; however, they need not be. For example, the names of the two items in the subroutine could equally well be C and D, in which case, of course, MOVE C to D would have been written with the same effect.

There are two methods of specifying which record in the Linkage Section is associated with which parameter address. Either the USING option of the Procedure Division clause may be used (see below) or, alternatively, the natural order of records in the Linkage Section defines the association. The second method means that the first record in the Linkage Section is taken as having the address of the first parameter in the CALL or ENTER, the second Linkage Section record as having the address of the second parameter and so on. If B preceded A in the subroutine's Linkage Section, the MOVE statement would act in the opposite direction from that intended; field B in the calling segment would be moved to

field A in the calling segment.

The Procedure Division USING clause

This clause is provided so that the user can, if he wishes, override the order of entries in the Linkage Section. The facility is particularly useful when splitting a large program into subroutines; one easy way to do this is to split the program into a main segment and a number of subroutines called from the main segment, and to reproduce the entire Working-Storage of the main segment under the Linkage Section of the subroutines.

Example

The original program might contain:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X1 PIC. 9.
.
.
.
01 Xn PIC 99.
PROCEDURE DIVISION.
FRED.
Paragraph body
JIM.
Paragraph body
BILL.
Paragraph body
STOP RUN.
****
```

The new program could be divided as follows:

Main segment

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X1 PIC 9.
.
.
.
01 Xn PIC 99.
PROCEDURE DIVISION.
FRED.
Paragraph body
CALL SUB1 USING X1, X2,....Xn
CALL SUB2 USING X1, X2,....Xn
STOP RUN.
****
```

SUB1

```
DATA DIVISION.
LINKAGE SECTION.
01 X1 PIC 9.
.
.
.
01 Xn PIC 99.
PROCEDURE DIVISION.
JIM.
Paragraph body
EXIT PROGRAM.
****
```

SUB2

Similar to SUB1

It will probably be true in practice that SUB1, for example, does not address all the records in the main segment's Data Division, but only, say, records X1, X5 and X7.

In this case, it would be possible to have only three records in SUB1's Linkage Section, say A1, A2 and A3 with identical descriptions to X1, X5 and X7 respectively (notice that in this case the names have been chosen to be different). Then the original CALL of SUB1 in the main segment would be replaced by

```
CALL SUB1 USING X1, X5, X7.
```

This would be much more efficient than the original CALL with many operands since it would generate only four machine code instructions. It is much more convenient for the user, however, to simply copy the original Data Division in his subroutine Linkage Section.

He can achieve this effect by writing in SUB1

```
LINKAGE SECTION.
01 X1
..  .
 .  .
 .  .
 .  .
01 Xn
PROCEDURE DIVISION USING X1, X5, X7
```

The CALL in the main program must now be written

```
CALL SUB1 USING X1, X5, X7.
```

Note that the number and order of parameters in the CALL statement correspond to the number and order of identifiers in the Procedure Division USING clause of the subroutine, if present. If the order of parameters in the CALL statement has been changed, as for example,

```
CALL SUB1 USING X7, X5, X1.
```

a corresponding change would have had to be made in the Procedure Division USING clause to

```
PROCEDURE DIVISION USING X7, X5, X1.
```

in order to preserve the intended association of parameter addresses with field names.

The EXIT PROGRAM statement

The only way of exiting from a subroutine is by means of the EXIT PROGRAM statement; in particular, dropping through the end of the subroutine does not cause exit from the subroutine but will cause the error halt, HALTED:- CZ to be displayed.

SPECIAL EFFECTS OF STANDARD COBOL OPTIONS

The ENTER verb

The ENTER verb may be used instead of CALL provided the parameters are restricted to the set that is legitimate with CALL. An important point should be noticed when using the ENTER verb to call PLAN subroutines. This is the fact that any field in the Linkage Section is treated in the same way as a field with a data name subscript. Thus, to load the address of such a field into X3 the code generated is not:

```
LDX 3 address
```

but:

```
CALL 0 address-calculation-routine
```

This means that if the user is accessing his parameters in the standard way, that is, by means of

```
LDX 1 link
OBEY 0 (1)
```

X0 will be overwritten. The user's PLAN subroutines should, therefore, be written in such a way as to take this into account.

The simplest way to avoid this problem is to extract all parameter addresses and store them in an area in Lower Storage before processing. It has been stated above that ENTER may be used instead of CALL to call COBOL subroutines provided that the parameters of the ENTER are record-names. However, it is possible to use ENTER in this way even where some of the parameters are not record-names provided that the parameters begin on word boundaries, and in the case of COMP and COMP-1 fields, are synchronized.

For example:

```
ENTER COBOL COBSUB USING A(B).
```

would be legitimate if A has as description:

```
02 A PIC 9 COMP SYNC RIGHT OCCURS 10.
```

The corresponding record in COBSUB's Linkage Section would have as its description:

```
77 A1 PIC 9 COMP SYNC RIGHT.
```

Note that if A had not been described as SYNC RIGHT the command would produce unpredictable results.

Users should note that where CALL can be used interchangeably with ENTER, the use of CALL will cause less code to be generated, especially if some of the parameters are in the Linkage Section.

The ACCEPT FROM and DISPLAY UPON statements

The user may use ACCEPT FROM and DISPLAY UPON statements with basic peripherals to which special names have been assigned in his subroutines, as well as in his main segment. The unit number is significant; for example, if in subroutine A, the user displays upon printer 3, and in subroutine B a file assigned to printer 3 is currently open, then the DISPLAY would display upon the same printer as is currently being used by subroutine B. The number of a peripheral unit from which information is accepted should not be the same as that assigned to a file in any other subroutine, since the ACCEPT statement releases the specified peripheral after execution.

The situation in both cases is exactly the same as if the file had been assigned within the same subroutine as the ACCEPT or the DISPLAY statement.

The STOP RUN statement

A STOP RUN statement may appear in a subroutine; in this case, however, the standard end of run checks will be performed on file closing, but only on those files defined in the sub-routine. Similarly, a STOP RUN statement in the main program will only check files defined in the main program.

ERROR MESSAGES

New and altered error messages are given below.

Message	*Cause*	*Action*
C17 TOO MANY NAMES ABANDON	There are too many library names or REPLACING parameters.	The program is abandoned.
065 CORR. ALONE	CORRESPONDING is out of context.	The statement being processed is ignored.
105 INVALID RECORDING MODE	As printed.	RECORDING MODE F is assumed.
125 TOO MANY DATA NAMES	More than one data-name follows, for example, REDEFINES or RENAMES.	Incorrect code is generated. Correct and recompile.
171 TOO MANY SUBROUTINES	The subroutine list is full.	All following subroutines are ignored.
172 S/R NAME > 11 CHARS	As printed.	The first 11 characters taken as the name.
173 (unused)		
183 INVALID MNEMONIC NAME	As printed.	The statement is ignored.
186 INVALID SWITCH CONDITION/ CONDITION NAME	The name used in a conditional statement is not a switch condition or a condition name.	The statement is ignored.
202 ONLY ONE LABEL IN GO DEPENDING	As written.	Control is passed to the next sentence for values >1
206 DEPENDING DATA-NAME INVALID	As written.	The maximum record size is assumed.
207 ENTER: TOO MANY ROUTINES INCLUDED	As in *COBOL*; the wording of the message is different)	

ICL International Computers Limited

User notice

MANUAL (NOTICE NO.) 23/12/70

4082 COBOL (58)

4078 COMPACT COBOL (34)

File one copy of this notice with each of the manuals indicated.

COBOL SUBROUTINES

This user notice supersedes User Notices COBOL (44) and Compact COBOL (26). There have been a number of minor changes and users are advised to see if they can now delete more subroutines from their libraries.

The list gives those subroutines on the COBOL M.L.T. that are applicable to each COBOL compiler. The list has been prepared chiefly for those users who wish to create their own program library tape or disc.

Reference should be made to the latest M.L.T. listing for the version numbers of the subroutines.

For those users who wish to be more economical in their selection, the subroutines have been classified according to six categories, any of which may be omitted if circumstances allow. Each category contains subroutines that may be omitted if one of the following is true:

1 No magnetic tape usage

2 No direct access usage

3 No RERUN statements

4 No direct access references by ENTERed subroutines (except for sorting)

5 No variable length field processing by ENTERed subroutine

6 No sorting

The compilers are given on the right-hand side (designated by the last two characters of the name); the presence of an asterisk in a compiler column indicates that a given subroutine may be applicable to that compiler.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

The subroutine categories are given on the left-hand side; the presence of an asterisk in a column indicates that a given subroutine will cease to be applicable if the condition for that category is true.

For example, %EROL may be applicable to compilers #XE18, #XE1B, #XE20, #XEKA and #XEKB (but not #XE11 or #XEA3) but ceases to be so if the condition 'no direct access usage' is true. This is indicated by an asterisk under column 2.

The list may contain references to subroutines or compilers not on the M.L.T. This is in order to avoid re-issue of the notice at frequent intervals. The references to software not on the M.L.T. can be ignored.

The subroutines in the list form the S-CM Subroutine Group. The S-CE Subroutine Group consists of all the subroutines marked as being applicable to the compilers #XE18, #XE1B, #XE20, #XEKA and #XEKB less those subroutines in category 1.

There is an incompatibility between the integral COBOL magnetic tape consolidators in #XE11 and #XE20 and other language subroutines of the following types:

1 All FORTRAN subroutines

2 All PLAN subroutines compiled by PLAN 4 compilers (#XPLN or #XPLT)

Magnetic tape users wishing to include subroutines of either of these types in a COBOL program should use #XEA3, #XE1B, #XEKA or #XEKB, if possible, or, if not, should use one of the above compilers for compilation only, and consolidate using the free standing consolidator #XPC1.

Any standard ICL subroutines that are *not* included in the COBOL M.L.T. should be assumed to be type 2 above and similar precaution should be taken.

The same remark applies to the subroutines %EROL, %FINROL, %AAINCORE and to both sets of GCBL subroutines: other subroutines included in the COBOL M.L.T. are compatible with any consolidator.

1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB
*						%TROL		*		*	*	*	*
	*					%EROL			*		*	*	*
	*					%FINROL					*		*
	*					%AAINCORE					*		*
	*					COBFIN					*		*
						CKBL							*
		*				CKBN							*
*		*				CKBO							*
						CKBQ							*
						CKBR							*
*						CKBRD							*
						CMBA							*
						CMBB							*
						CMBC						*	*
*		*				CMBE							*
	*	*				CMBF							*
	*	*				CMBG							*
						CMBL						*	
		*				CMBN						*	
						CMBO						*	*
						CMBS						*	*
		*				COBN				*	*		
						COBF				*	*	*	*
						COBS				*	*		
						COBA		*	*	*	*	*	*
						COBB		*	*	*	*	*	*
						COBM				*	*	*	*
						COBO				*	*		
						COBP				*	*	*	*
						COBQ				*	*	*	
						COBR				*	*	*	
						COBZ				*	*	*	*
						COBI		*	*	*	*	*	*
						COBJ		*	*	*	*	*	*
						COBL				*	*		
						COBY		*	*				

1	2	3	4	5	6	#XE	11	18	A3	1B	20	KA/KB

1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB
						COBC		*	*				
						COBK		*	*				
						COBH		*	*				
						COBG		*	*				
						COBD		*	*				
*		*				COBE			*				
*		*				COBX			*				
*		*				COBV		*					
*					*	COBOLSORT		*	*	*	*	*	*
*					*	CMSORT		*	*	*	*	*	*
*					*	CMSORTA		*	*	*	*	*	*
*					*	CMSORTB		*	*	*	*	*	*
*					*	CMSORTC		*	*	*	*	*	*
*					*	CMSORTD		*	*	*	*	*	*
*					*	CMSORTE		*	*	*	*	*	*
*					*	GCBLMTSORT		*	*	*	*	*	*
*					*	GCBLMTSORT1		*	*	*	*	*	*
*					*	GCBLMTSORT2		*	*	*	*	*	*
*					*	GCBLMTSORT3		*	*	*	*	*	*
*					*	GCBLMTMAIN		*	*	*	*	*	*
*					*	GCBLMTDUMP		*	*	*	*	*	*
*					*	GCBLMTAPSE		*	*	*	*	*	*
*					*	GCBLMTLODE		*	*	*	*	*	*
*					*	GCBLMTCLOD		*	*	*	*	*	*
*					*	GCBLMTDASE		*	*	*	*	*	*
*					*	GCBLMTDALD		*	*	*	*	*	*
*					*	GCBLMTINTS		*	*	*	*	*	*
*					*	GCBLMTOUTS		*	*	*	*	*	*
*					*	CQ5R5S5							*
	*				*	CSORT		*	*	*	*	*	*
	*				*	CSORTA		*	*	*	*	*	*
	*				*	CSORTB		*	*	*	*	*	*
	*				*	CSORTC		*	*	*	*	*	*
	*				*	CSORTD		*	*	*	*	*	*
	*				*	CSORTE		*	*	*	*	*	*
					*	CPERM		*	*	*	*	*	*
	*				*	GCBLDASORT		*	*	*	*	*	*
1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB

COBOL (58)
COMPACT COBOL (34)

1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB
	*				*	GCBLDASORT1	*	*	*	*	*	*	
	*				*	GCBLDASORT2	*	*	*	*	*	*	
	*				*	GCBLDASORT3	*	*	*	*	*	*	
	*				*	GCBLHSKPNG	*	*	*	*	*	*	
	*				*	GCBLCOBDUMP	*	*	*	*	*	*	
	*				*	GCBLMTSURCH	*	*	*	*	*	*	
	*				*	GCBLMTLOAD	*	*	*	*	*	*	
	*				*	GCBLDASURCH	*	*	*	*	*	*	
	*				*	GCBLDALOAD	*	*	*	*	*	*	
	*				*	GCBLCOBOLLD	*	*	*	*	*	*	
	*				*	GCBLINBATCH	*	*	*	*	*	*	
	*				*	GCBLOUTBTCH	*	*	*	*	*	*	
	*				*	CQ1R1S1						*	
	*				*	CQ2R2S2						*	
				*		COBPAK	*	*	*	*	*	*	
				*		COBPAKM	*	*	*	*	*	*	
				*		COBDIST	*	*	*	*	*	*	
				*		COBDISTM	*	*	*	*	*	*	
				*		PUNCHN	*	*	*	*	*	*	
	*		*			COPENINPUT	*	*	*	*	*	*	
	*		*			COPENOUTPUT	*	*	*	*	*	*	
	*		*			COPENEND	*	*	*	*	*	*	
	*		*			COPENINOUT	*	*	*	*	*	*	
	*		*			COPSCRATCH	*	*	*	*	*	*	
	*		*			CALLOPENS	*	*	*	*	*	*	
	*		*			CSREADA	*	*	*	*	*	*	
	*		*			CSINSERTA	*	*	*	*	*	*	
	*		*			CSDELETEA	*	*	*	*	*	*	
	*		*			CFILENOSET	*	*	*	*	*	*	
	*		*			CREADINDEX	*	*	*	*	*	*	
	*		*			CSREAD	*	*	*	*	*	*	
	*		*			CSINSERT	*	*	*	*	*	*	
	*		*			CSDELETE	*	*	*	*	*	*	
	*		*			CUSEINDEX	*	*	*	*	*	*	
	*		*			CRREAD	*	*	*	*	*	*	
	*		*			CRINSERT	*	*	*	*	*	*	
1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB

1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB
	*		*			CRDELETE	*	*	*	*	*	*
	*		*			CRECBUCPO	*	*	*	*	*	*
	*		*			CREAD	*	*	*	*	*	*
	*		*			CWRITE	*	*	*	*	*	*
	*		*			CREPLACE	*	*	*	*	*	*
	*		*			CDELETE	*	*	*	*	*	*
	*		*			CALLWRITE	*	*	*	*	*	*
	*		*			CACTION	*	*	*	*	*	*
	*		*			CCLOSE	*	*	*	*	*	*
	*		*			CRESET	*	*	*	*	*	*
	*		*			CRESCLO	*	*	*	*	*	*
	*		*			CRETURN	*	*	*	*	*	*
	*		*			CREPEAT	*	*	*	*	*	*
	*		*			CEXCANAL	*	*	*	*	*	*
	*		*			CEXTEND	*	*	*	*	*	*
	*		*			CCONTRACT	*	*	*	*	*	*
	*		*			CEXT	*	*	*	*	*	*
	*		*			CLEVEL1	*	*	*	*	*	*
	*		*			CLEVEL2	*	*	*	*	*	*
						CDTA	*	*	*	*	*	*
						CDTB	*	*	*	*	*	*
						CDTC	*	*	*	*	*	*
						COBDTANDX	*	*	*	*	*	*
*		*				HDMT	*	*	*	*	*	*
		*				HDUMP	*	*	*	*	*	*
	*	*				HDEDS					*	*
	*	*				HDFDS					*	*
	*	*				HDDAT					*	*
*						HMTRRB						*
*						HMTT						*
*						HMTEND	*	*	*	*	*	*
*						HMTWR	*	*	*	*	*	*
*						HMTCRE			*	*	*	*
*		*				HMTCLB	*	*	*	*	*	*
*						HMTRD	*	*	*	*	*	*
*						HMTM	*	*	*	*	*	*
1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB

1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB
*						HMTN		*	*	*	*	*	*
*						HMTP		*	*	*	*	*	*
*						HMTH							*
*						HMTO		*	*	*	*	*	*
*						HMTQ		*	*	*	*	*	*
*						HMTC		*	*	*	*	*	*
*						HMTE		*	*	*	*	*	*
*						HMTD		*	*	*	*	*	*
*						HMTF		*	*	*	*	*	*
*						HMTHA							*
*						HMTAA		*	*	*	*	*	*
*						HMTMOD		*	*	*	*	*	*
*						HMTDEF		*	*	*	*	*	*
	*					HRAMOD		*	*	*	*	*	*
	*					HRADEF		*	*	*	*	*	*
	*					HRARD		*	*	*	*	*	*
	*					HRADEL		*	*	*	*	*	*
	*					HRAWR		*	*	*	*	*	*
	*					HRAEXT		*	*	*	*	*	*
	*					HRAWRI		*	*	*	*	*	*
	*					HRACLB		*	*	*	*	*	*
	*					HRAEND		*	*	*	*	*	*
	*					HRAOVR		*	*	*	*	*	*
	*					HRATGS		*	*	*	*	*	*
	*					HRAOF		*	*	*	*	*	*
	*					HRACOF		*	*	*	*	*	*
	*					HRAN		*	*	*	*	*	*
	*					HRAA		*	*	*	*	*	*
	*					HRAR		*	*	*	*	*	*
	*					HRAG		*	*	*	*	*	*
	*					HRAM		*	*	*	*	*	*
	*					HRAIND		*	*	*	*	*	*
1	2	3	4	5	6		#XE	11	18	A3	1B	20	KA/KB

© International Computers Limited, Reading, 1970

International Computers Limited

User notice

MANUAL (NOTICE NO.) 23/12/70

4082 COBOL (57)

File one copy of this notice with each of the manuals indicated.

COBOL COMPILERS #XEKA/3, #XEKB/3

The information in this notice supplements that given in the general description in User Notice COBOL (46) and in Amendment List 6 to *COBOL*.

The Mark 3 issue of the compilers #XEKA and #XEKB has been enhanced as follows:

1 Magnetic tape work files may be specified for the SORT statement as an alternative to direct access files.

2 The NO REWIND and RETAIN phrases may be used in the CLOSE statement.

3 The REVERSED phrase may be used in the OPEN statement.

4 The DECIMAL POINT IS COMMA clause may be specified in the SPECIAL-NAMES paragraph.

5 In the COPY statement (see page 107 of *COBOL*) words in the COBOL library may also be replaced by identifiers, and hardware-names may replace and be replaced although they are reserved words. The format of the COPY statement is now as follows:

COPY library-name [REPLACING word-1 BY
$\left\{\begin{matrix}\text{word-2}\\\text{identifier-1}\end{matrix}\right\}$ [word-3 BY $\left\{\begin{matrix}\text{word-4}\\\text{identifier-2}\end{matrix}\right\}$] ...] .

where library-name is composed according to the rules for COBOL data-names (see page 15 of *COBOL*), and word-*n* is a hardware-name or any valid COBOL word that is not a reserved word; that is it may be a data-name, procedure-name, file-name, mnemonic-name, hardware-name or literal. Word-1 word-3 etc. are in the library entries and are to be replaced, in the output source, by word-2 or identifier-1, word-4 or identifier-2 etc.

FORM 1/230/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

6 COBOL Procedure Division sections can be written in any order regardless of priority; further, sections with the same priority need not be adjacent. When taking advantage of this facility for an object program on magnetic tape it is necessary to do a separate consolidation with #XPC1.

7 The duplication of standard constants and scratch pad in programs containing COBOL subroutines is reduced to 19 words. The remaining words will be catered for in a block of COMMON Lower Preset, called COBLP, 31 words in size, and a block of COMMON Lower Variable called COBLV, up to 42 words in size. If no COBLV area is required by a program the cue will not be present; COBLP will always be present.

8 The generation of one PLAN segment per COBOL section in a master program. This will ease the problem that exists where the segments generated are too big to fit in store with the cue-list during the loading process. Note, however, that the cue-list will be longer in programs containing several segments because of the inter-segment jumps.

9 Implementation of the steering line:

*APPEND EDS (file-name)

This causes the compiler to pass an APPEND parameter to #XPCK and #XPCL; this parameter is to define a composite file to which the binary object program is to be appended. In its implementation if the output file is not a composite file the consolidator creates one; if it is a composite file it replaces the end-of-composite-file bucket with a start-of-subfile bucket, appends the subfile and then writes an end-of-composite-file bucket.

New error message

Message	*Cause*	*Action*
649 DISORDERED SEGMENTS - NO INTEGRAL CONSOLIDATION	An attempt has been made to enter the integral consolidator when the program contained disordered segments.	The run is abandoned.

Code list: reference number definitions for error messages 006, 011 and 012

0081 _SECTION

0100 See 0102 and note on page 237 of *COBOL*.

0102 _ {OF / IN} 100

0119 See 0123 and note on page 237 of *COBOL*.

0123 (119 ...)

0131 ACCEPT_

0144 ACCEPT FILE KEYS FROM_

0148 ACCEPT_FROM_

0160 _ACTUAL

0174 ADD CORR_TO_

0183 ADD CORR_TO_SIZE ERROR 0257

0186 ADD CORR_TO_ROUNDED SIZE ERROR 0257

0191 ADD CORR_TO_ROUNDED

0194 See 0198, 0203, 0209, 0215 and note on page 237 of *COBOL*.

0198 ADD 0194 GIVING_

0203 ADD 0194 GIVING_ROUNDED

0209 ADD 0194 GIVING_ROUNDED SIZE ERROR_ 0256

0215 ADD 0194 GIVING _SIZE ERROR_ 0256

0218 See 0224, 0233, 0239, 0245 and note on page 237 of *COBOL*.

0221 See 0224, 0233, 0239, 0245 and note on page 237 of *COBOL*.

0224 ADD {0218 TO 0221 / 0228 0227}

0227 See 0224, 0233, 0239, 0245 and note on page 237 of *COBOL*.

0228 See 0224, 0233, 0239, 0245 and note on page 237 of *COBOL*.

0233 ADD {0218 TO 0221 / 0228 0227} ROUNDED

0239 ADD {0218 TO 0221 / 0228 0227} ROUNDED SIZE ERROR 025_

0245 ADD {0218 TO 0221 / 0228 0227} SIZE ERROR 0257

0257 See 0183, 0186, 0209, 0215, 0239, 0245, 0605, 0658, 0664, 0675, 0681, 1050, 1056, 1067, 1073, 1428, 1434, 1440, 1446, 1551, 1560, 1783, 1786, 1809, 1815, 1826, 1832.

0263 ALL_

0272 _ALPHABETIC

0281 ALTER_TO PROCEED TO_[_TO PROCEED TO_]

0296 ....OR....AND_ in an abbreviated condition (Type 2)

0307 _AND_ in an abbreviated condition (Type 2)

0318 ......UNEQUAL TO ..... AND_

0322 _AND_

0324 _AND_

0333 See 0342 and note on page 237 of *COBOL*.

0340 See 0342 and note on page 237 of *COBOL*.

0342 APPLY 0333 [TO] SORT-* [ON] 0340

0350 APPLY_SENTINEL-PROCESSING [ON] _

0354 See 0365 and note on page 237 of *COBOL*.

0363 See 0365 and note on page 237 of *COBOL*.

0365 APPLY 0354 [TO] INDEX-BUFFER [ON] 0363

0373 APPLY ____ [TO] FILE-MESSAGES [ON] _

0376 See 0385 and note on page 237 of *COBOL*.

0383 See 0385 and note on page 237 of *COBOL*.

0385 APPLY 0376 [TO] TRANSFER-REPLY [ON] 0383

0407 ASSIGN OBJECT_PROGRAM [TO] _

0420 BIT-*nn* [0433] [0444]

0433 ON STATUS IS_ see 0420 and note on page 237 of *COBOL*.

0444 OFF STATUS IS_ see 0420 and note on page 237 of *COBOL*.

0462 BLOCK [_TO] _RECORDS

0469 BLOCK [_TO] _CHARACTERS

0478 CALL_

0482 See 0484 and note on page 237 of *COBOL*.

0484 CALL_ USING 0482

0489 CARD_PUNCH_ [1876]

0496 CARD_READER_ [1876]

0502 CHANNEL-N_

0511 { CHECK / CHEQUE } PROTECT LEAVING_

0517 CLOSE_

0520 _REEL

0526 _NOREWIND

0527 _REWIND

0599 DELETE_

0605 DELETE_INVALID [KEY] 0257

0617 DISPLAY_

0620 See 0623 and note on page 237 of *COBOL*.

0623 DISPLAY 0620 UPON_

0642 DIVIDE_INTO_

0647 DIVIDE_INTO_GIVING_

0652 DIVIDE_INTO_GIVING_ROUNDED

0658 DIVIDE_INTO_GIVING_ROUNDED SIZE ERROR 0257

0664 DIVIDE_INTO_GIVING_SIZE ERROR 0257

0669 DIVIDE_INTO_ROUNDED

0675 DIVIDE_INTO_ROUNDED SIZE ERROR 0257

0681 DIVIDE_INTO_SIZE ERROR 0257

0686 EDS_

0698 ENTER_ _

0796 See 0671 and note on page 237 of *COBOL*.

0708 ENTER_USING 0669

0730 _ { EQUALS / EQUAL TO / = } _

0754 EXAMINE_TALLYING _ _

0751 EXAMINE_TALLYING _ _ REPLACING BY _

0777 EXAMINE_REPLACING _ _ BY _

0788 FDS_

0796 FILE_RECORDS

0816 FLOAT SIGN LEAVING_PLACES

0836 See 0797 and note on page 237 of *COBOL*.

0842 GO 0797 DEPENDING ON_

0854 - { GREATER THAN / EXCEEDS / > } -

0884 IF (condition) 0860 [{ ELSE / OTHERWISE } 0860 ...

0960 See 0044 and note on page 237 of *COBOL*.

0909 INPUT ----

0917 INPUT_REVERSED

0918 INPUT-OUTPUT_

0934 _IS_

0967 - { LESS THAN / < } -

0983 _LOCK

0992 MCF_

1000 MEMORY (SIZE)_

1003 MEMORY (SIZE) _WORDS

1013 MOVE CORR_TO_

1018 See 0975 and note on page 237 of *COBOL*.

1020 MOVE_TO 0973

1034 MULTIPLY_BY_

1039 MULTIPLY_BY_GIVING

1044 MULTIPLY_BY_GIVING_ROUNDED

1050 MULTIPLY_BY_GIVING ROUNDED SIZE ERROR 0256

1056 MULTIPLY_BY_GIVING_SIZE ERROR 0256

```
1061   MULTIPLY_BY_ROUNDED
1067   MULTIPLY_BY_ROUNDED SIZE ERROR 0256
1073   MULTIPLY_BY_SIZE ERROR 0256
1077   _NEGATIVE
1089   _NO REWIND
1101   NOT_
1105   NOT_
1112   _NUMERIC
1119   OBJECT-COMPUTER_
1125   OCCURS_
1130   OCCURS_TIMES
1134   OCCURS_ [TIMES] DEPENDING_
1147   OPEN_
1151   OPEN [1156] [1161]
1156   INPUT_...   }
1161   OUTPUT_...  }   See 1151 and note on page 237 of COBOL.
1172   _OR_        }   in an abbreviated condition (Type 2)
1184   ... UNEQUAL TO ... OR_
1193   _OR_
1240   OUTPUT_ ---
1243   OUTPUT_ ----- AT END
1248   PAPER_PUNCH_ [1876]
1255   PAPER_READER_ [1876]
1270   PERFORM_ [THRU_]
1282   PERFORM_ [THRU_] _TIMES
1294   PERFORM_  THRU_ UNTIL_
 318   {VARYING} _FROM_BY_UNTIL_ See 1321 and note on page 237 of COBOL.
       {AFTER  }
1321   1318 [1318] ...   See 1324 and note on page 237 of COBOL.
1324   PERFORM_  THRU_  VARYING 1321
1331   PICTURE_
1339   POINT LEFT_
1344   POINT RIGHT_
1347   POINT LEFT_BITS
1350   POINT RIGHT_BITS
1354   _POSITIVE
1359   PRINTER_ 1876
1395   PROCEDURE DIVISION [USING] 1403
1403   See 1395 and note on page 237 of COBOL.
1416   RANGE_
1428   READ_INTO AT END 0257
```

1434 READ_INTO_INVALID 0257

1440 READ_AT END 0257

1446 READ_INVALID 0257

1454 RECORD [_TO] _ [CHARACTERS]

1461
1464 RECORDING MODE { F / V / 2 }

1479 REDEFINED_

1485 RELEASE_

1489 RELEASE_FROM_

1502 RENAMES_

1508 RENAMES_THRU_

1519 RERUN ON { FDS-* / MCF-* / TAPE-* } - REEL_

1523 RERUN ON { FDS-* / MCF-* / TAPE-* } _ RECORDS_

1527 RERUN REEL_

1531 RERUN_RECORDS

1537 RESERVE_ [ALTERNATE AREA]

1551 RETURN [RECORD] - AT END 0257

1560 RETURN [RECORD] - INTO_ AT END 0257

1567 REWRITE_

1571 REWRITE_FROM_

1580 SAME AREA_

1584 SAME RECORD AREA_

1594 _SECTION [_]

1607 SEEK_ [RECORD]

1611 SEGMENT_LIMIT_

1623 See 1625 and note on page 237 of *COBOL*.

1625 SELECT_ASSIGN_ [1623]

1627 SELECT_ASSIGN_

1630 SELECT_ASSIGN_ (Select)

1635 1630 [1630 ...]

1650 SIZE_ [CHARACTERS]

1653 SIZE_DIGITS

1663 ASCENDING_ ...

1669 DESCENDING_ ...

1674 { 1663 / 1669 } [{ 1663 ... / 1669 ... }] See 1694, 1707, 1712, 1717, and note on page 237 of *COBOL*.

1694 SORT_ 1674 INPUT PROCEDURE 1728 OUTPUT PROCEDURE 1728

1707 SORT_ 1674 USING_OUTPUT PROCEDURE 1728

1712 SORT_ 1674 INPUT PROCEDURE 1728 GIVING_

1717 SORT_ 1674 USING_GIVING_

1728 See 1694, 1712

1734 SOURCE-COMPUTER_

1751 STOP_

1769 SUBTRACT CORR_FROM_

1774 SUBTRACT CORR_FROM_ROUNDED

1783 SUBTRACT CORR_FROM_SIZE ERROR 0257

1786 SUBTRACT CORR_FROM_ROUNDED SIZE ERROR 0257

1789 See 1793, 1798, 1808, 1809, 1815, 1820, 1826, 1832 and note on page 237 of *COBOL*.

1793 SUBTRACT 1781 FROM_

1798 SUBTRACT 1781 FROM_GIVING_

1808 SUBTRACT 1781 FROM_GIVING_ROUNDED

1809 SUBTRACT 1781 FROM_GIVING_ROUNDED SIZE ERROR 0257

1815 SUBTRACT 1781 FROM_GIVING_SIZE ERROR 0257

1820 SUBTRACT 1781 FROM_ROUNDED

1826 SUBTRACT 1781 FROM_ROUNDED SIZE ERROR 0257

1832 SUBTRACT 1781 FROM_SIZE ERROR 0257

1837 SYMBOLIC KEY_

1865 _THRU_

1876 TYPE_

1886 _UNEQUAL TO_

1907 VALUE_...

1916 VALUE OF {ID / IDENTIFICATION} _

1920 See 1923, 1928 and note on page 237 of *COBOL*.

1923 VALUE OF {ID / IDENTIFICATION} _ACTIVE-TIME 1920

1926 See 1928, and note on page 237 of *COBOL*.

1928 VALUE OF {ID / IDENTIFICATION} _ACTIVE-TIME 1920
GENERATION-NO 1926

1931 See 1933 and note on page 237 of *COBOL*.

1933 VALUE OF {ID / IDENTIFICATION} _GENERATION-NO 1931

1944 WRITE_

1949 WRITE_FROM_

1955 WRITE_INVALID 0257

1961 WRITE_FROM_INVALID 0257

1965 WRITE_FROM_BEFORE_

1969 WRITE_FROM_AFTER

1973 WRITE_BEFORE_

1977 WRITE_AFTER_

1994 ZERO SUPPRESS LEAVING_

2933 SAME SORT AREA_----

2940 DATA RECORDS_ ------

Corrections

The errors reported under the following Commercial Languages Software Notice item numbers have been corrected:

32, 33, 34, 42, 51, 56, 65, 66, 67, 68, 70, 71, 72, 79, 80, 81, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 95, 96, 97, 98, 99, 100, 101, 103, 104, 105, 109, 111,113,114,115, 116, 117, 118.

© International Computers Limited, Reading, 1970

International Computers Limited

User notice

MANUAL (NOTICE NO.)

2/9/70

4082 COBOL (55)

File one copy of this notice with each of the manuals indicated.

*OVERLAY AND ENTERED SUBROUTINES

This User Notice replaces the item **OVERLAY and ENTERed subroutines* in User Notice COBOL (41) of 1/4/70, and should be read in conjunction with User Notice COBOL (23), *COBOL subroutine facilities*.

The *OVERLAY steering line (which can be used only when compiling master programs) instructs the consolidator to include in each overlay any subroutine ENTERed or CALLed in the overlay, provided the subroutine is not ENTERed or CALLed in any other overlay or in permanent. However users should note that there are possibilities of error when subroutines themselves call other subroutines, or when they contain cues (which can happen only with subroutines not written in COBOL).

If, for instance, a subroutine is indirectly called, for example, subroutine A ENTERed in the permanent part of the program in turn calls subroutine B which has been overlaid by means of *OVERLAY, there is no guarantee that at run time, when subroutine B is called, it will be in store, unless the user ensures that it is by programming. This situation could occur not merely with user's PLAN or COBOL subroutines, but when using the COBOL direct access subroutines. For example, if CREAD is ENTERed in one overlay, then it will be included in that overlay (if *OVE IS used). However, CSREAD, CSREADA and CRREAD all contain branches to CREAD and so if any of these three subroutines are ENTERed in permanent or in a different overlay from CREAD, the results at run time may be unpredictable. In the same way:

CSINSERT
CSINSERTA
CRINSERT
} contain branches to CWRITE

and

CSDELETE
CSDELETEA
CRDELETE
} contain branches to CDELETE

FORM 1/230/45(3.69)

The solution in this situation is to write a dummy ENTER CREAD (or CWRITE or CDELETE) in, for example, the permanent part of the program or in a special, separate overlay thus forcing the subroutine into permanent. As the ENTER will never be obeyed, it need not have any parameters.

A problem may also arise with cue names. The compiler cannot distinguish between segment names and cue names, both of which may be ENTERed. Thus the consolidator may be told that a cue name occurs in one overlay, although the segment name containing the cue is referenced in a different overlay. Alternatively two different cues in the same subroutine, such as GCBLDASORT4 and GCBLDASORT5, may be ENTERed in different overlays. In this case consolidation would fail, with unpredictable results. The solution is *either* to insert dummy ENTERs of GCBLDASORT4 and GCBLDASORT5 in permanent or in different overlays from those of the original ENTERs *or* to ensure that both cues are ENTERed in the same overlay together with a dummy ENTER GCBLINBATCH (the segment in which the cues are contained), and that all three are referenced in that overlay only.

The user should also remember, in this connection, that

1 GCBLDASORT6 is a cue within GCBLOUTBTCH

2 GCBLMTSORT4 and GCBLMTSORT5 are cues within GCBLMTINTS

3 GCBLMTSORT6 is a cue within GCBLMTOUTS.

} in the magnetic tape sort subroutines

© International Computers Limited, Reading, 1970.

International Computers Limited

User notice

MANUAL (NOTICE NO.) 26/8/70

4082 COBOL (52)

File one copy of this notice with each of the manuals indicated.

COBOL COMPILER XEKA/2A

The information in this notice supplements that given in the general description User Notice COBOL(46).

As this first version of XEKA is closely related to XEKB/2A this new compiler is shortly to be issued as XEKA/2A.

In addition to the facilities described in COBOL(46) this version will allow, by treating the sequence as noise words, the SAME SORT AREA clause in the I-O-CONTROL paragraph.

COBOL COMPILER XEKB/2A

The information in this notice supplements that given in the general description User Notice COBOL(46).

In addition to the facilities described in COBOL(46) this version will allow, by treating the sequence as noise words, the SAME SORT AREA clause in the I-O-CONTROL paragraph.

Corrections

The errors reported under the following Commercial Languages Software Notice item numbers have been corrected:

46,47,48,50,52,53,54,55,57,58,59,60,61,62.

Alters in Notice item number 63 should not be applied to XEKB/2A.

SORTING MAGNETIC TAPE FILES BY SUBROUTINE

Introduction

FORM 1/230/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

A set of PLAN subroutines is now provided to enable users to write COBOL programs sorting files held on magnetic tape. Communication is by way of the ENTER statement in the Procedure Division. The system used is the same as the one described in *COBOL* manual, page 160, for sorting files held on direct access devices. The names of the corresponding routines are given below.

The parts of the system

1 THE USER ENTERED ROUTINES

CMSORT
CMSORTA
CMSORTB
CMSORTC
CMSORTD
CMSORTE

2 THE CORE-RESIDENT ROUTINES

GCBLMTSORT	GCBLMTAPSE
GCBLMTSORT1	GCBLMTLODE
GCBLMTSORT2	GCBLMTDASE
GCBLMTSORT3	GCBLMTDALD
GCBLMTMAIN	GCBLMTCLOD
GCBLMTDUMP	GCBLMTINTS
	GCBLMTOUTS

These subroutines should be put into the subroutine group (or groups) which will be searched by the COBOL compiler in question. Group 1 must precede group 2.

Of the group 2 routines one of the first four (GCBLMTSORT, GCBLMTSORT1, GCBLMTSORT2, GCBLMTSORT3) and one or both of the last 2 (GCBLMTINTS, GCBLMTOUTS) which control input and output procedures will be incorporated into the COBOL program for each sort depending which group 1 routine is entered. The other seven will always be incorporated.

Note that CPERM is not included among group 1; it is no longer necessary to use it, except to prevent an unsatisfied cue to %ROL in a non-overlay program containing procedures. (Users who are not familiar with CPERM will find a description of it in the *COBOL* manual, page 160.4).

3 THE SORT PROPER, #XSXM

#XSXM is a non-overlay program which is required to reside on a library file (MT or direct access).

When the group 2 routines are entered via group 1, COBOL object program is dumped to the work files and #XSXM is read in from its library file as if it were coming from the COBOL program's overlay file; when the sort is finished, the COBOL program is reloaded and entered accordingly.

General

1 The subroutines will sort a complete input file written in Housekeeping formats on a single reel of magnetic tape terminated by an end of reel or end of file sentinel.

2 Data will be output to a complete file, consisting of one or more reels written in Housekeeping formats.

3 At least three, but not more than ten, scratch tapes (as specified by a parameter) are used: Naturally, the more scratch tapes are used the more efficient the sort will be.

4 Instead of using an input or output file the user may present records one by one to the sort as input and on output the sorted records may be presented to the user one by one. To do this he must supply an input and/or output procedure. The following combinations of input and output are available:

Input file and output file: see CMSORTA

Input procedure and output file: see CMSORTB

Input file and output procedure: see CMSORTC

Input procedure and output procedure: see CMSORTD

Keys

A description of keys is given in the *COBOL* manual, page 160.1.

Files

All files to be used by the sort must be closed on entry to it and they will be left closed on exit, but while a procedure is being obeyed one work file will be needed by the subroutine associated with it.

INPUT FILE

The subroutines will sort an input file written in Housekeeping formats on one reel of magnetic tape. A write-permit ring is unnecessary, but if there is not one present and the file has no retention period it may be picked up (e.g. as one of the scratch tapes during the sort). Data will be terminated by an End-of-Reel or End-of-File Sentinel.

Any information which is held on the input tape either between the Header Label and the start of Data Sentinel or between Dump Sentinels will be ignored so long as this information does not include a tape mark. The data block counts held in User Sentinels and the Trailer Label will be checked but other information held in these and in the start of Data Sentinel will be discarded and will not appear in the output file.

Record format

Records may be either fixed or variable in length up to a maximum of 4095 words. The first word of each record must contain, in the least significant fifteen bits, a binary count of the number of words, including itself, in the record. This word will not be altered in any way by the sort.

Block format

A data block may contain one or more complete records and its maximum size will be that specified by a parameter. (A data block may be neither greater than 4095 in length nor shorter than 5.

OUTPUT FILE

Reel format

The output data will be written to a scratch tape which will be relabelled according to the parameter specified in CMSORTA/E. Word 1 of the start of Data Sentinel will contain the maximum block size, and Word 1 of the Trailer Label the number of data blocks on the reel. The contents of words 4-19 of the Qualifier Blocks are unspecified. If, as a result of new blocking arrangements or a short output tape, the data cannot all be written on one reel, continuation reels will be used until a reel sequence number of 511 is reached.

Record format

The record format on an output file is the same as for an input file, no portion of any record being altered in any way.

Block format

An output block will contain one or more complete records which will be close packed at the beginning of the block. For an output file the maximum block length must be specified by the user. However, the actual maximum may be less than this, and a smaller figure will be written in the Start-of-data sentinel if there is not enough core available to allocate buffers of the necessary size.

During the merge no blocks are written which are shorter than 20 words, but on the last phase the minimum block size is reduced to 5 words. Any block containing fewer than 5 words of data will be expanded with zero filler words.

WORK FILES

The sort uses from 3 to 10 work files which have expired retention periods. As each is opened it is relabelled SCRATCH∇TAPE and retains this header at the end of the sort. Before the sort is entered the user's program is written to 2 work files. This duplication is necessary because one of the files containing the dumped program may become the output file at the last phase. It should be noted that it is the core allocated to the user's program at run time which is written to the work files, not necessarily the compiled allocation, e.g. with a single programming executive the whole core store will be written.

If the user has specified an output procedure, one work file is used as the output file. In this case the first reel is labelled ICL∇OUTREELA and given a retention period of 1 day. Continuation reels have file names ICL∇OUTREELB, ICL∇OUTREELC and so on. When the merge has finished and the user's program has been reloaded data is read from the first reel and records presented one by one to the output procedure. When a reel is exhausted it is relabelled SCRATCH∇TAPE and closed.

Calling the subroutines

NOTES ON PARAMETERS

Wherever a generation number or a reel sequence number appears as a parameter to a subroutine it must be the name of a field with PICTURE S1(23) or PICTURE 1(24), SYNCHRONIZED RIGHT respectively. This field is expected to contain the required generation number or reel sequence number on entry to the sort. (Its value may be ACCEPTed etc.). If the former field is positive, the twelve least significant bits are taken as the generation number. If a generation number is zero for a magnetic tape file or negative for a direct access file, no check will be made.

Procedure-name may be the name of a procedure in an overlay of the object program. This may be a different overlay from the one containing the sort.

File-name, for files described in the Data Division, refers to the name in the SELECT...ASSIGN statement.

CMSORT: THE SORT ENVIRONMENT.

ENTER PLAN CMSORT USING *file-name, fgn, device-type, data-name-1, data-name-2.*

File-name is an alphanumeric literal (or zero), being the name of the library file containing #XSXM; *fgn* is its generation number and *device-type* gives the device on which it resides, as follows:

1 = E.D.S., 2 = twin E.D.S., 3 = F.D.S.,

4 = Magnetic tape.

If *file-name* is zero then the next two parameters are ignored (but must be present) and the library file is assumed to be PROGRAM XSXM, generation number zero, resident on magnetic tape.

Data-name-1 is either zero or the name of a field described by PICTURE 1(24) which contains the maximum block size of the input file in words. This parameter must be present (even if there is no input file, in which case it will be ignored); if it is zero a maximum input block size of 200 words will be assumed.

Data-name-2 is identical to *data-name-1* except that it refers to the output file.

CMSORT must be used whenever a change in environment is required, it being unnecessary to call the sub-routine again if a sort has the same environment as the last one. However, initially (i.e. before the first sort) the default options given above are assumed, so that if these are not appropriate CMSORT must be called to change them as required.

CMSORTA: SORT WITH INPUT FILE AND OUTPUT FILE

ENTER PLAN CMSORTA USING *file-name-1, rsn, fgn-1, file-name-2, fgn-2, record-name*

File-name-1, rsn and *fgn-1* are the name, reel sequence number and generation number of the magnetic tape input file described in the Data Division.

File-name-2 and *fgn-2* are the name and generation number of the output magnetic tape file described in the Data Division.

Record-name is the name of a record described in Working-Storage Section. It has the following contents:

Words 1-2	Zero
Word 3	The length in binary words of the procedure buffer. If there are no procedures, this word will be zero. If the word is zero and there is a procedure then a buffer of at least 256 words is assumed.
Word 4	The number of work tapes (at least 3 but not greater than 10) to be used by the sort.
Words 5-7	Zero (the sort will use word 6 as a reply word)
Word 8	The number of keys (in binary).

The rest of the area is of variable length, there being a 2-word entry for each key. These 2 words must hold:

Word 1	Character 0	1-6 = type of key
	Character 1	0 = ascending key
		1 = descending key

	Characters 2 and 3	Length of key in binary, in bits if key is type 1 or 2, in characters if key is type 3-6.
Word 2	Start address of key.	
	Key type 1 or 2:	B0-4 bit address B12-23 word address
	Key type 3-6:	B0-1 character address B12-23 word address

CMSORTB: SORT WITH INPUT PROCEDURE AND OUTPUT FILE

ENTER PLAN CMSORTB USING *procedure-name, file-name, fgn, record-name, area-1, area-2.*

Procedure-name is the name of the procedure that is used to present records to the sort. (*Section name* or *paragraph name* in the Procedure Division).

File-name and *fgn* are the name and generation number of the output magnetic tape file described in the Data Division.

Record-name is as in CMSORTA.

Area-1 is the name of the area in which the input procedure sets up records. It must begin on a word boundary.

Area-2 is the name of an area to be used as a buffer during the input procedure. Its size is described in Word 3 of record-name.

CMSORTC: SORT WITH INPUT FILE AND OUTPUT PROCEDURE

ENTER PLAN CMSORTC USING *file-name, rsn, fgn, procedure-name, record-name, area-1, area-2.*

File-name, rsn and *fgn* are the name, reel sequence number and generation number of the input magnetic tape file described in the Data Division.

Procedure-name is the name of the procedure which accepts records from the sort.

Record-name is as for CMSORTA

Area-1 is the name of the area in which records are presented to the user within the output procedure. It must begin on a word boundary.

Area-2 is as for CMSORTB except that it is used during the output procedure.

CMSORTD: SORT WITH INPUT PROCEDURE AND OUTPUT PROCEDURE

ENTER PLAN CMSORTD USING *procedure-name-1, procedure-name-2, record-name, area-1, area-2.*

Procedure-name-1 is as for *procedure-name* in CMSORTB.

Procedure-name-2 is as for *procedure-name* in CMSORTC.

Record-name is as for CMSORTA.

Area-1 is used for both purposes described in CMSORTB and CMSORTC.

Area-2 is as for CMSORTB and CMSORTC.

CMSORTE: TO RETURN TO THE COBOL STATEMENT FOLLOWING THE SORT.

ENTER PLAN CMSORTE.

Use of this routine will cause control to be passed to the COBOL statement following the last executed CMSORTA/B/C/D. It will normally be used in the output procedure when it is not desired to process the sorted file through to the end. CMSORTE must not be used to return control to a different overlay.

WITHIN INPUT AND OUTPUT PROCEDURES

1 GCBLMTSORT4: To present a record to the sort during an input procedure.

ENTER PLAN GCBLMTSORT4

When the input procedure has set up a record in area-1, this routine is used to present it to the sort.

2 GCBLMTSORT5: To enter the sort at the end of the input procedure.

ENTER PLAN GCBLMTSORT5

When all the records have been presented to the sort this routine is used to enter the sort proper.

3 GCBLMTSORT6: To obtain a record from the sort during an output procedure.

ENTER PLAN GCBLMTSORT6 USING *procedure-name*.

This routine will present the next record of the sorted file to the user (in *area-1*).

Procedure-name is the name of the procedure to be branched to when the end of the file is reached.

Note GCBLMTSORT4/5 are cues within GCBLMTINTS.
GCBLMTSORT6 is a cue within GCBLMTOUTS.

Exception conditions

All exception conditions found by the sort will be indicated by a message on the console typewriter. This message will consist of the words SORT HALT or SORT ERROR followed by a two-character code. SORT HALT implies that the program may be able to continue after operator action. SORT ERROR means that the program cannot continue.

The error codes are listed below in alphabetical order.

Code	*Reason*	*Action*
CE	Checksum failure	Abandon run and restart at beginning. If error persists report for investigation.
E1	Number of keys is not between 1 and 10.	Correct and rerun.

Code	*Reason*	*Action*
E2	Key type not in range 1-6	Correct and rerun.
E3	The key direction is not 0 or 1.	Correct and rerun.
E4	Key length is invalid.	Correct and rerun.
E5	Error in key address.	Correct and rerun.
R1	The sort needs more core.	Give program more core and GO.
R2	There is not enough core to reload the user's program.	Give program more core and GO.
R4	#XSXM is not on the library file.	Correct and rerun.
S1	A record has a zero word count.	Correct and rerun.
S2	The input procedure has presented more records than can be held on a work tape.	Correct and rerun.
S3	The sort cannot use the given number of work tapes.	Correct and rerun.
S4	Error in Input File Format.	Correct and rerun.
S5	The sequence number of an output reel has exceeded 511.	Correct and rerun.
S6	Record length error.	Correct and rerun.
S7	Record counts do not balance.	Correct and rerun.
S8	There has been a failure in the merge buffering system.	Report for investigation.
S9	Short work tape.	Abandon run.
T0	The user has used an output procedure after entry 29 (see below) has been used to re-load the COBOL program.	Abandon run.

Code	*Reason*	*Action*
T1	Parity error.	Abandon run. If error persists report for investigation.
T2	Checksum error when reading library.	Abandon run. If error persists report for investigation.
T4	Checksum failure when reloading COBOL program.	Abandon run. If error persists report for investigation.
T7	Data specifying block not found on library	Abandon run. Correct library format and rerun.
T8	Entry block found in the wrong place on the library file.	
T9	Tape mark found in the wrong place.	

The first character of an error code gives an indication of the action which should, or may, be taken. In the case of the code CE and those codes whose first character is T there is no way of continuing the run; the job should be abandoned and, if possible, re-run when the cause of the error has been corrected. If a code begins with either E or S, the run may not be continued except by using entry 29. Error codes whose first character is R indicate that the sort may be resumed after operator action but that entry point 29 may not be used.

Notes

It is not permitted to use either the VALUE OF ID IS *data-name*, or the ACTIVE-TIME IS *data-name* facility in the FD description for either the input or output file to the sort (when present). (The output file is opened in output mode during the sort, and will be given a retention period according to the ACTIVE-TIME clause.)

The input and output file (if present) must not be open on entry to the sort.

The sort will search for two unit numbers which are not in use (at time of entry) by the COBOL program, and will open its files under these numbers. The method of searching for unit numbers will be to start at 15 and work downwards. All files opened by the

sort will be closed again before control is returned to the COBOL program unless an output procedure is present, when one file will be retained until termination of the sort.

With the above in mind the user must ensure that no files with unit numbers used by the sort are opened within input and/or output procedures.

The sort requires a minimum of 5888 words of core store. If this much is not allocated to the COBOL object program, the sort will attempt to allocate itself the extra core needed. If 5888 words are not available, then the message SORT ERROR R1 will be produced on the console. It should be stated that while 5888 words is the minimum amount of core in which the sort can ever function, it will in fact need an amount not less than (1400+M) X F words, where M is the maximum input block size and F the number of work files.

RESTART POINT 29

After any error which has a code beginning with E or S the COBOL program may be reloaded by typing the message GO#XXXX29 (where XXXX is the name of the COBOL program). Control is returned to the COBOL program accordingly. In this case word 6 of record-name will be set to 1. In all other cases this word will be zero.

CONSOLE MESSAGES

During a sort the following messages may be output on the console:

A When a work tape is no longer required, the message

DISPLAY:WORK FILE CLOSED ON UNIT NN

is typed, NN indicating the deck number of tape concerned.

Similarly the message

DISPLAY:INPUT FILE CLOSED ON UNIT NN

is produced when all the data has been read from an input file.

B When the output phase of the merge is reached, a scratch tape will be opened and renamed as the first reel of the output file. At this point the sort is suspended and the program types

HALTED: LOAD OUTPUT FILE

The appropriate scratch tape should be loaded and the sort continued.

C If the sorted data should fill an output reel, this reel is closed, and the messages

DISPLAY : OUTPUT REEL CLOSED ON UNIT NN

DISPLAY : LOAD CONTINUATION REEL

are output; separated, if there is an output procedure, by

DISPLAY; PLEASE PRESERVE THIS REEL

In this case the deck will not be disengaged and the tape should be retained, to be re-opened for use by the routine which presents records to the output procedure.

D When the final reel of the output file is closed this is indicated by the message

DISPLAY OUTPUT FILE CLOSED ON UNIT NN

E At the end of each sort the number of records is indicated by

DISPLAY : NNNNN RECORDS SORTED

Example program with input procedure and output file

THE PROBLEM

The records of a master file held on E.D.S. contain two keys, a name and a job-number. Before processing, the records are held in ascending order of job-numbers and alphabetical order of names within each job-number. It is required to do some processing of the file whilst in this order and then to sort the records into alphabetical order of names and descending order of job-number within each name, and write them to an output file on magnetic tape.

An input procedure will be used to present the records to the sort.

The sort is in PROGRAMVXSXM, generation number 10 on magnetic tape.

Four work tapes will be used including one for the input procedure.

THE RELEVANT COBOL STATEMENTS

```
1    Environment Division.

     SELECT UNSORTED-FILE ASSIGN E.D.S. 1
     ACCESS MODE SEQUENTIAL.
     SELECT SORTED-FILE ASSIGN 1 TAPES.

2    Data Division.

     FILE SECTION.

     FD  UNSORTED-FILE LABEL RECORDS STANDARD

         VALUE OF ID "USF".
     01  RECORD-1.
     02  COUNT PICTURE 9(4) COMPUTATIONAL SYNCHRONIZED
         RIGHT.
     02  JOB-NO PICTURE 9(5).
     02  NAME PICTURE X(13).

     FD  SORTED-FILE LABEL RECORDS STANDARD VALUE
         OF ID "SF".

     01  RECORD-2.
     02  COUNT2 PICTURE 9(4) COMPUTATIONAL SYNCHRONIZED
         RIGHT.
     02  JOB-NO2 PICTURE 9(5).
     02  NAME2 PICTURE X(13).

     WORKING-STORAGE SECTION

     01  REC.
     02  WDS1-2 PICTURE 9(8) VALUE ZEROS.
     02  WD3 PICTURE 9(4) VALUE 128.
     02  WD4 PICTURE 9(4) VALUE 4.
     02  WD5 PICTURE 9(4) VALUE ZERO.
     02  WD6 PICTURE 9(4) VALUE ZERO.
     02  WD7 PICTURE 9(4) VALUE ZERO.
     02  WD8 PICTURE 9(4) VALUE 2.
     02  KEY1.
     03  PT1A PICTURE 99 VALUE 30.
     03  PT1B PICTURE 1(12) VALUE 13.
     03  PT2A PICTURE 1(2) VALUE 1.
     03  PT2B PICTURE 1(22) VALUE 2.
     02  KEY2.
     03  PART-1 PICTURE 9(4) VALUE 4105.
```

```
    03  PART-2 PICTURE 9(4) VALUE 1.
    01  BUFFER.
    02  DESC PICTURE X(64) OCCURS 8 TIMES.
    77  FGN PICTURE S1(23) SYNCHRONIZED RIGHT
        VALUE 10.
    77  FGN2 PICTURE S1(23) SYNCHRONIZED RIGHT
        VALUE 20.
    77  OUBL PIC 1(24) VALUE 200.

3   Procedure Division.

    BEGIN.

        OPEN INPUT UNSORTED-FILE.
        ENTER PLAN CMSORT USING "PROGRAM XSXM" FGN.
        4 0 OUBL.
        ENTER PLAN CMSORTB USING INPROC SORTED-FILE.
        FGN2 REC RECORD-1 BUFFER.
        CLOSE UNSORTED FILE.
        STOP RUN.

    INPROC.

        READ UNSORTED-FILE AT END GO TO TERM.
        Process record.
        ENTER PLAN GCBLMTSORT4.
        GO TO INPROC.

    TERM.

        ENTER PLAN GCBLMTSORT5.
```

© International Computers Limited, Reading, 1970

MANUAL (NOTICE NO.) 1/7/70

4082 COBOL (50)

File one copy of this notice with each of the manuals indicated.

#XE1B

This user notice, together with Amendment List 4 to *COBOL*, supersedes the previous information on #XE1B in User Notices COBOL 28, pages 5 to 13, and COBOL 38, pages 3 to 6. Pages 1 to 4 of User Notice 28 should already have been discarded in accordance with an instruction given in User Notice 33 and the information on pages 13 to 23 is contained in Appendix 2 to User Notice 46; User Notice 28 can thus be discarded. Similarly, pages 1 to 3 of User Notice 38 have been superseded by User Notice 39 and so User Notice 38 can also be discarded.

FORM 1/230/45(3.69)

LANGUAGE FACILITIES

The language facilities are described fully in *COBOL*. In particular Part 1, Chapter 1, Table 2 lists the available language facilities and User Notice COBOL 23 gives additional information on the subroutine aspects. It should be noted however, that the following are not available.

```
RERUN
JUSTIFIED RIGHT
VALUE OF ID IS     data-name
ACTIVE-TIME IS     data-name
GENERATION-NO IS   data-name
CORE-MT
SELECT ... ASSIGN TAPES n
```

Programs generated by #XE1B differ from those generated by #XE20 and #XEA3 in a few respects and details are given below. In general code generated for statements in subroutines is the same as that generated for similar statements in a master program.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

Ambiguities of messages

The run time message LOAD FILE *nn* ON *pp* may be ambiguous in object programs using subroutines since *nn* refers to the *nn*th file in the Data Division of the COBOL subroutine issuing the message. If this matters to the user, he can get over the problem by inserting a DISPLAY statement before the relevant OPEN statement to inform the operator which file is indicated.

The run time error message for files may also be ambiguous, as, for example, READ AFTER END OF FILE SENT *nnnnnn* where *nnnnnn* is the sequence number within the subroutine issuing the message.

For ACCEPT statements referring to the console typewriter, the standard COBOL ACCEPT (SEE JOB SHEET) ... message is preceded by the message DISPLAY - COBOL SUBROUTINE *subroutine-name*, if the ACCEPT statement was written in a subroutine. This is because the sequence number quoted in the message might not otherwise be sufficient to identify the message uniquely.

Compiler listings

When #XE1B is compiling a program the listings are identical to those produced by #XEA3 and #XE20. When compiling a subroutine the listings are the same except for fields in the Linkage Section. These are given addresses relative to the start of their parent 01 level.

For example

```
01    A
02    B   PIC   X   (7)
02    C   PIC   X   (2)
```

would be mapped as

```
MAP  01  A                    0.0.0.
MAP  02  B                    0.0.0.
MAP  02  C                    1.3.0
```

If A is in the Linkage Section.

The cue list from the consolidators exhibits some differences both for programs and for subroutines.

1 PROGRAMS The only difference from the programs compiled by #XEA3 and #XE20 relates to the relativizers LP, LV, UP and UV, whose addresses appear at the head of the consolidated cue list. For #XEA3 and #XE20 the number printed after these headings indicates the start address of the object program's Lower Preset, Lower Variable, Upper Preset and Upper Variable areas respectively.

For programs compiled by #XE1B, however, these relativizers are not used. The corresponding data areas are assigned to the common areas %LV, %LP, %UP

and %UV respectively; these names will occur in the cue list and their addresses are printed next to them.

Users will have noticed that in the cue list output for a COBOL object program pure numeric cues, for example, 3, 47 and 101, occur; for subroutines compiled by #XE1B these are transformed by appending the subroutine name, to make them unique; for example, if the subroutine name is SUBR, they would appear as 3SUBR, 47SUBR and 101SUBR. These numeric cues are, however, suppressed by the integral COBOL consolidator and only printed by free-standing consolidators. These cues may greatly increase the size of the consolidator's cue table and indeed may make it impossible to consolidate a program including many subroutines. For this reason, users are recommended to make the names of subroutines as short as possible even though up to 11 characters are permitted.

2 SUBROUTINES Subroutine data is allocated under Lower Preset and Literals. The consolidated cue list does not print out the addresses of these relativizers for each subroutine, but only the beginning address for the whole area. On the other hand, if in the compiler map a field's address, for example, is given as 100, 0, 0 UP, this means word 100 of this particular subroutine's Upper Preset. The only way of finding out the address of these areas is by inspection of the object code; the beginning of the program area for the current subroutines can be found from the subroutine cue, that is, the name given in the PROGRAM-ID clause, with type 41 or 45 in the cue list.

Overlay programs

The CORE-EDS facility does not affect the meaning of the SEGMENT-LIMIT clause, or of the priority number 50. Thus, as with any normal overlaying system, segments with priority number less than the segment limit go into the permanent part of the program; segments with priority number greater than 50 are brought down in their initial state while other overlay segments are brought down in their last used state. However, there is an important qualification to the last statement, in that it applies only to sections of the master program; any subroutine included in any overlay unit of a master program is brought down in its initial state whenever the overlay unit in which it is included is physically reloaded, that is, in practice, whenever there has been a reference to some different overlay unit since the last reference to the current one. Otherwise, the subroutine remains in its last used state. These comments apply not only to ALTERed GO TO statements but also to the subroutine's data areas since these are included in the same overlay unit as the subroutine itself. It is therefore the user's responsibility to ensure correct initialization of subroutine data areas. (This remark applies to all overlay systems and not merely to %FINROL)

The CORE-EDS facility requires the inclusion of the overlay subroutines, %FINROL, %AAINCORE and COBFIN.

For master programs written in COBOL, these routines are included automatically, if required. For master programs in PLAN, the user should write

```
#PERMANENT %FINROL
```

in his steering segment; it is not necessary to mention %AAINCORE.

The user may also define in a PLAN segment a lower common location, %AALIMIT and preset it to a value n where $1<n<1023$; this will have the effect that, for each overlay area, units 1 to $n-1$ inclusive are held in core and units n to 1023 on backing store. %AALIMIT need not be defined, in which case the action is similar with n taken as 501.

If the master program is in COBOL, %AALIMIT is automatically preset to 25.

Error messages

The following additions have been made to the code list reference numbers for #XE1B, which are otherwise the same as the list in *COBOL*, pages 234.3 to 237.

Reference	*Type of sentence containing invalid material*
2055	CALL subroutine-name
2078	PROCEDURE DIVISION USING
2089	See 2091
2091	CALL subroutine-name USING 2089
2117	ASSIGN OBJECT-PROGRAM TO (peripheral)
2121	FDS
2125	EDS

The following messages have been added to the error message list:

1 761 ILLEGAL PARAMETERS

This is printed when any of the parameters mentioned in the USING clause of the Procedure Division header are not record-names. The erroneous parameters are ignored.

2 759 ILLEGAL LINKAGE SECTION

The Linkage Section occurs in a master program; the Linkage Section entries are counted as if they were Working-Storage entries.

3 799 ILLEGAL SUBPROUTINE PARAMETER

One of the parameters of a CALL statement is not a record-name, that is, a 01 or 77 level. The parameter is ignored.

4 827 FIELD NOT PARAMETER

A subroutine is referencing a field name in the Linkage Section whose parent record name has not been declared in the USING clause of the Procedure Division header. The field is given an arbitrary address.

Note: This message may appear several times for the same statement.

User subroutine limits

A maximum of 50 distinct user subroutine names may be referred to in a program or subroutine compiled by #XE1B.

Not more than 18 parameters may be written as operands to a CALL or ENTER statement.

ENVIRONMENTAL FACILITIES

Minimum configuration

The minimum configuration for compilation is:

12032 words of core store
1 80-column card reader or 1 paper tape reader
2 x 2801 or 2 x 2802 E.D.S. transports or 2 x 2820 or 2 x 2821 twin-E.D.S. transports or 4 magnetic tape decks,
1 line printer

When used in a disc-only environment this compiler is designed for use with two transports: this applies equally to twin-E.D.S. and E.D.S. However, the compiler will, subject to restrictions in speed and capacity, run on one transport. The compiler can be found in more core store than the minimum. The only advantage in doing so is that it will allow the compiler to compile very large source programs and/or reduce the amount of data written to the workfiles. The maximum number of words the compiler should be given is 31,744.

When #XE1B resides on twin-E.D.S. or E.D.S. the file containing it should be at least 1520 blocks in size.

Steering lines

Where files accessed by the compiler are to be on E.D.S. the operand of the steering line has the format:

EDS (file details)

The file details are the same as for magnetic tape except that the reel number and retention period cannot be used (see pages 185 to 187 of *COBOL*). The subfile name can be used only for the *OBJECT file, see below. All other options, including renaming, can be used. Note that the omission of a generation number will cause the default option, highest on-line, to be used.

The compiler does not distinguish between twin-E.D.S. and E.D.S. All reference to E.D.S. in this user notice is to be taken to include twin-E.D.S. Twin-E.D.S. is identified in the steering lines as EDS.

Where files accessed by the compiler are to be on magnetic tape the operand can, in a number of cases, include a subfile name. Details are given in the relevant sections below.

Workfiles

The compiler's workfiles are assumed to be on the same medium as the compiler itself unless either switch 21 is set, in which case workfiles are assumed to be on magnetic tape, or switch 23 is set, in which case workfiles are assumed to be on E.D.S. Three workfiles are required if the workfile medium is magnetic tape. The file named in the *OBJECT steering line will constitute one of these if it is on magnetic tape. The files named in the *COBOL or *MASTER steering line will also constitute workfiles when they are on magnetic tape and M has not been written in the steering line. The compiler will use the spare tape, beyond the end-of-composite file trailer label, as workspace. If M is not written and the tape is loaded without a write permit ring, the compiler will go ILLEGAL trying to write to the tape.

Four workfiles are required if the workfile medium is E.D.S. and may be one of three possible types:

1 A user named file

2 A conventionally named file, that is, COBVWORKFILE

3 A scratch file

User named workfiles are declared in the steering lines by the directives:

```
*WK1
*WK2
*WK3
*WK4
```

The directives may be presented to the compiler in any order. Only one of each directive may be present. Any or all of the directives may be omitted. The opening of each workfile is performed as described. If a *WK*n* directive is present, the named file is opened. If the file cannot be opened the message with reference number 677 is typed at the console and a scratch file is opened.

For the workfiles that are not specified in a *WK*n* directive an attempt is made to open a file named COBVWORKFILE. The first such file must have generation number 1, the second generation number 2, and so on. There is no relationship between the *n* of *WK*n* and the generation numbers of the COBVWORKFILE files. If a COBVWORKFILE file cannot be opened, a scratch file is opened. The name COBVWORKFILE with any generation number can appear in a *WK*n* directive. For

example, if *WK1 and *WK3 steering lines specified user named files, the compiler would expect to use COBVWORKFILE, generation 1, as the second workfile and COBVWORKFILE, generation 2, as the fourth workfile. Further if, say COBVWORKFILE generation 1 was not available a scratch file would be opened instead. Thus in this case the four workfiles would be:

1 File named in *WK1

2 Scratch file

3 File named in *WK3

4 COBVWORKFILE generation 2

The first two workfiles must be allocated size 160 blocks, or 8 blocks per 20 lines of source COBOL compiled using them whichever is the greater. The third workfile must be allocated size 8 blocks per 20 lines of source COBOL compiled using it.

The fourth workfile is used during compilation to contain details of the literals written in the Constant Section and Working-storage Section of the COBOL source program. The size of the file is therefore a function of the number and size of the literals in the source program. Each literal occupies 5 words, plus the number of words required to contain the literal, and the number of words available to contain information in each block is 126. A file of 80 blocks is therefore sufficient to contain 1680 literals of 4 characters, or 240 of 128 characters.

Source programs

The *MASTER and *COBOL files can each be on magnetic tape or E.D.S. irrespective of the medium containing the compiler or its workfiles.

Where the source program files are on E.D.S. they should be allocated the number of blocks required to contain one source program plus one block for a subfile description. If the source program lines have columns 73 to 80 punched, each block of a source program file will contain six COBOL lines. If the identification columns are not punched, complete words of spaces after the rightmost significant character on each line are not recorded on the file.

Where the files are on magnetic tape #XE1B will read a COBOL source program contained in a subfile of a single or multiple subfile composite file. File and subfile generation numbers may be quoted in the steering line and will be checked. Omission of the subfile name will cause #XE1B to search for a subfile named COBOLVVV*aaaa*, where *aaaa* is the four-character name from the *IDENTITY steering line. Omission of the file or subfile generation number will cause no check to be made.

#XE1B will write a COBOL source program in subfile format to a magnetic tape file named in a *COBOL steering line or will

add a new subfile to the end of an existing file. For the former the subfile name and generation number may be quoted in the steering line. If the subfile name is not quoted, the compiler will write it as COBOLVVV*aaaa*, where *aaaa* is the four-character name from the *IDENTITY steering line. If the subfile generation number is not quoted it will be written as zero. To add a subfile to the end of an existing file the *COBOL steering line operand must take the form:

FILENAME = . SUBFILENAME

The subfile generation number is optional, as above.

Users are warned that if the . is accidentally omitted from a steering line operand such as:

NAMEONE = .NAMETWO

it becomes

NAMEONE = NAMETWO

The effect of this, of course, is to overwrite the file NAMEONE instead of adding data to the end of it. The file is also relabelled NAMETWO.

Compilation of programs or subroutines without consolidation

*OBJECT MT

The effect of the *OBJECT steering line is similar to that for source, except if the =. facility is used. If no subfile name is quoted the subfile will be given the name PROGRAMV*aaaa* where *aaaa* is the four character name given in the *IDE steering line.

The =. facility in the *OBJECT steering line has the meaning described when a subroutine is being compiled. If the subfile name quoted is the same as that of the last subfile on the tape, the current subroutine is appended to the last subfile; if the subfile name is different, a new subfile is opened at the end of the tape with the quoted subfile name, and the current subroutine put into it.

In the case of programs, however, a new subfile will be opened in any case, possibly with the same name as the previous subfile.

The warning about omitting the . from an =. steering line, given under source programs, applies equally here.

Suppose the user is compiling four COBOL subroutines in succession, named SUBR1 and SUBR2, SUBR3, and SUBR4, which are all to be compiled on to a tape named FRED in two subfiles named SUB1 and SUB2 respectively, the *OBJECT lines for the four successive compilations would be:

```
*OBJ    MT    (SCRATCH TAPE = FRED.SUB1)
*OBJ    MT    (FRED = .SUB1)
*OBJ    MT    (FRED = .SUB2)
*OBJ    MT    (FRED = .SUB2)
```

Assuming that the compilations were in the order SUBR1, SUBR2, SUBR3, and SUBR4, the first subfile in the file labelled FRED would be SUB1 which would contain the two subroutines SUBR1 and SUBR2 and the second (and last) subfile would be SUB2, which would contain the two subroutines SUBR3 and SUBR4.

Normally the user will wish to have only one subfile per output tape, especially since the free-standing consolidator #XPC1 always examines the whole of a file and does not recognise individual subfiles.

#XE1B is not a batch compiler, and therefore the facility of creating a tape containing a number of different subroutines will involve a separate compilation for each.

Normally the user will not wish to hold a group of semi-compiled programs on an output tape, but only a group of subroutines. A group of programs may, however, be placed on a single tape; they must be in unconsolidated semi-compiled form, that is, the *CONSOLIDATE steering line may not be used for any of the compilations.

*OBJECT EDS

The effect of the *OBJECT steering line for an E.D.S. file is similar to that for a MT file except for the effect of the =. facility. This differs from the corresponding facility on magnetic tape in that it is not possible to add a new subfile of semicompiled program but only to add semi-compiled to an existing subfile.

Suppose, therefore, that it is required to compile into an E.D.S. file called SEMICOMPILED and consolidate a program named FRED with two subroutines SUB1 and SUB2. The relevant steering lines for the three runs are:

```
1    *IDE    FRED
     *OBJ    EDS (SEMICOMPILED)
     [*OVE]

2    *IDE    FRED
     *LIN    .
     *OBJ    EDS (SEMICOMPILED=.PROGRAM FRED)

3    *IDE    FRED
     *LIN
     *OBJ    EDS (SEMICOMPILED=.PROGRAM FRED)
     *CON
```

The subfile name given to the object file is always PROGRAMV*aaaa*, where *aaaa* is the four-character name from the *IDE steering line of the program or subroutine currently being compiled. Thus if *IDE BERT were quoted for the second compilation above it would be necessary to

quote *OBJ EDS (SEMICOMPILED=.PROGRAM BERT) in the third compilation, since the new subfile name replaces the old one.

If the subfile named in the steering line cannot be found or has the wrong format, the message:

627 OBJECT FILE HAS WRONG FORMAT

will be printed and compilation will be abandoned.

Note: When processing a file created by means of this facility #XPCK expects to find the master program before any subroutines. Thus the user is recommended to compile the master program before the subroutines if possible, otherwise #XPCK will assume the default name #XXXX for the object program. If the master program is compiled first and the name in the *IDE steering line for the last subroutine is different from that for the master program, #XPCK will take the program name from the master program. However, if the *BINARY steering line was present, it will halt CONSOLIDATED #*name* where *name* came from the *IDE steering line of the last subroutine. If this matters to the user, he should ensure that the same name is quoted in the *IDE steering line for all segments of the program.

The *OBJECT file can also be on cards or paper tape when consolidation is not required in the same run.

Compilation of programs and their consolidation with #XE1B

The *OBJECT file can be on magnetic tape or E.D.S. The effect of *CONSOLIDATE and *SUBROUTINES is given below.

*OBJECT MT

The presence of a *CONSOLIDATE steering line will cause the integral consolidation phase of the compiler to be entered. A *OVERLAY steering line cannot be used with integral consolidation. If the compiler is resident on magnetic tape, any subroutines required for consolidation will be expected to follow the compiler on the same library tape. An additional subroutine file may, however, be specified in a *SUBROUTINES steering line, and this file can be on cards, paper tape, or magnetic tape, but not E.D.S. In this event, if any subroutines are required the file named in the *SUB steering line is scanned first, and the library tape on which the compiler is resident is then scanned only if more subroutines are required.

If the compiler is resident on E.D.S. there is no library file of subroutines on E.D.S. and a *SUB steering line must be supplied. The medium can be cards, paper tape or magnetic tape (but not E.D.S.) and the operand will quote the name of the file containing the compiler's run time subroutines. User subroutines quoted in ENTER statements must also be held in this file.

*OBJECT EDS

The presence of a *CONSOLIDATE steering line will cause the compiler to delete itself after the compilation phase and load #XPCK. *CON MT causes the message

DELETED: FI #XPCK #TAPE

to be typed at the end of compilation, and *CON EDS causes the message

DELETED: FI #XPCK

to be typed.

No file name may be specified in the *CON steering line, and if the medium (MT or E.D.S) is omitted, the peripheral on which the compiler resides is assumed.

If no *SUBROUTINES steering line is supplied, the consolidator will expect to find a subroutine file on E.D.S. with the name SUBROUTINES.

If *SUB EDS (file name) is supplied, the consolidator will look for the required subroutines in the quoted file, and will not look for them in the standard file called SUBROUTINES.

The *OBJECT file should be allocated with approximately 8 blocks per 30 lines of source COBOL, plus any space needed for subroutines to be copied to the end of it.

#XPCK required a user named workfile. This is provided by #XE1B naming to #XPCK the first workfile that it used, that is, the one named in a *WK1 steering line or, in its absence, COBVWORKFILE, generation 1. The automatic consolidation facility cannot be used if the first of the #XE1B workfiles was a scratch file or if the workfiles were on magnetic tape.

Separate consolidation of object programs

This is required when the *OVERLAY steering line is used and the object program is to be on magnetic tape. For this, consolidation has to be done with #XPC1. Details can be found in COBOL User Notice 48 Appendix 2.

Where the user requires to present more than one subroutine file for consolidation he must use #XPC1 or #XPCK, supplying the necessary parameters on cards or paper tape.

*BINARY steering line

The presence of this steering line will cause #XPCK to write the binary object program to the E.D.S. file named in the operand. With this steering line only the file name and generation number can appear in the operand.

The file should be allocated with the name PROGRAMV*xxxx*, where *xxxx* are any four characters, the first being alphabetic. The four characters need not be the same as the name of the object program the file is to contain. However, the file will only contain one binary program at a time. The size of the file is a function of the size of the largest object program it is to contain, and is in the ratio of 8 blocks to 50 lines of source COBOL.

Extension and contraction of E.D.S. files

If the compiler completely fills an output file during the run it will attempt to extend the file by 80 blocks and continue writing. This may happen at any time during compilation and may happen to more than one file and/or more than once to a particular file. The extensions will be on any available cartridge.

The actual size of any single extension may be greater than 80 blocks, depending on the configuration of the unallocated space on the cartridge. Prior to terminating the compilation the compiler will contract any of the workfiles that were extended, back to their original size.

SPECIAL NOTE

Where a correction run ended prematurely, or contained AMENDMENTS OUT OF SEQUENCE errors the resulting *COBOL file will be corrupt. If this file is presented as input to a subsequent correction run, the run will fail, probably with the compiler looping. When the *COBOL file for the second run is on E.D.S. the loop will cause the file to be extended over all available space on the cartridges that are on-line. When the *COBOL file for the second run is on magnetic tape the tape will appear to run away. To avoid this the user should make the appropriate alterations to his amendment lines and repeat the first correction run. In any case, this is the recommended practice.

E.D.S. usage

All files used by the compiler (other than scratch files) must be allocated by #XJEC (see the ICL 1900 Series manual *Library Specifications* (Edition 2, TP4011) for details). The files must contain one block per bucket.

The only important time-saving consideration in file mapping is to ensure that the first two workfiles are on different cartridges. If this is not possible, they should be allocated so that one of them occupies blocks 0 to 39 of a set of contiguous cylinders and the other occupies blocks 40 to 79 of the same set of cylinders. Further savings can be achieved by applying similar rules to the *OBJECT and the *BINARY files, the *OBJECT and the *SUBROUTINES files, and the *MASTER and the *COBOL files.

The only other file required for a complete compilation system is XPEBOOTSTRAP. Its size is 16 blocks. Its use is explained in *Library Specification* under #XPEB.

© International Computers Limited, Reading, 1970

ICL International Computers Limited

User notice

19/5/71

MANUAL (NOTICE NO.)

4082 COBOL (62)

File one copy of this notice with each of the manuals indicated.

A version of the COBOL compiler will be available shortly which will incorporate the facilities described in User Notice *COBOL* (59), pages 2 to 7. The increase in the maximum number of digits/bits allowed in arithmetic operation will affect the information given on rules of arithmetic as described in Chapter 9 of *COBOL*.

This user notice describes the rules of arithmetic that will apply when using the new arithmetic facilities. This information replaces Chapter 9 of *COBOL* and it will be incorporated into a new edition of this manual in due course. Meanwhile however, it must be remembered that the current Chapter 9 applies to all programs not written for compilation by the new compiler.

RULES OF ARITHMETIC

Introduction

There are five arithmetic verbs, viz, ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE. The formats of statements involving them are given on pages 74, 102, 91 and 82 respectively, of *COBOL*, and in User Notice *COBOL* (59) concerning the COMPUTE enhancements.

MOVE is also considered an arithmetic verb in the cases where one of its operands is a decimal field or literal and the other is a binary field since arithmetic conversion occurs in these cases (see page 7 below).

Definitions

The *point position* of any quantity is considered to be positive (+N places) if the quantity has point left N places and negative (-N places) if it has point right N places.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

The *result* of a computation is simply the numeric result of the computation at run time.

The *receiving field* is the field in which the result is finally stored.

Basic concepts

The following comments are intended only as a general outline. Modifications and variations due to the individual verbs and the classes of their operands will be considered later in greater detail. For purposes of explanation it is assumed that the result and the receiving field have the same class.

1 ROUNDING If after decimal point alignment the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction in the receiving field, truncation to the size of the receiving field takes place. However, if ROUNDED is specified, the absolute value of the receiving field is increased by 1 whenever the most significant digit of the excess is greater than or equal to half the radix of that place. When integer positions in a receiving field are represented by the character P in the picture for that receiving field, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated. If no truncation could possibly take place and ROUNDED has been specified, the compiler outputs the message ROUNDING CLAUSE POINTLESS.

2 SIZE ERROR If, after decimal point alginment, the value of a result exceeds the largest value that can be contained in its specified receiving field, a size error condition exists. When such a size error condition occurs, the subsequent action of the compiler depends on whether or not the SIZE ERROR option has been specified.

If SIZE ERROR has not been specified and a size error condition occurs, the contents of the specified receiving field will be unpredictable.

If SIZE ERROR has been specified and a size error condition occurs, the contents of the receiving field will remain unchanged and the SIZE ERROR clause will be obeyed.

3 THE COMPOSITE The composite may be considered as the result of a computation in which all the operands hold their maximum possible values. If such a result is too large for the compiler to deal with correctly, the message COMPSITE TOO BIG is output. The action taken by the compiler on encountering this condition depends on the particular case.

In general, if a COMPOSITE TOO BIG message has been signalled, run time results that exceed the maximum permitte composite size will not be detected by a SIZE ERROR test. Results of such computations may therefore be unpredictable even when a SIZE ERROR condition has been specified.

4 NEGATIVE RESULTS

(a) Rounding is always performed on the modulus of the result, that is, the absolute value of the result, ignoring whether it is signed; for example, -3.5 is rounded to -4 and not to -3.

(b) If the result of a computation is negative and the receiving field is unsigned, the modulus or absolute value of the result will be placed in the receiving field and the sign will be ignored; for example, -5185 would be stored as 5185.

5 INTERMEDIATE RESULTS Intermediate results can occur in 3 types of statements.

(a) ADD/SUBTRACT with multiple operands after the verb

(b) COMPUTE statements specifying a series of arithmetic operations

(c) Arithmetic expressions in IF and PERFORM statements

The compiler treats the statement as a series of operations. Consider the following example.

```
COMPUTE R = (A+B)*(C+D)/(E-F**G)
```

This is broken down by the compiler into

```
ADD      A    B          GIVING IR1
ADD      C    D          GIVING IR2
MULTIPLY IR1  IR2        GIVING IR3
**F   BY  G              GIVING IR4
SUBTRACT IR4  FROM E     GIVING IR5
DIVIDE   IR5  INTO IR3   GIVING R
```

Where IR1, IR2 ... are intermediate results, which may be store locations or accumulators. Successive intermediate results may use the same locations.

The maximum permissible size of an intermediate result is 30 digits if the operands are both decimal or sterling and 100 bits if the operands are both COMP-1.

If these limits are exceeded for statements of type (a) above the message CAUTION COMPOSITE TOO BIG is printed. If the limit is exceeded for statements of types (b) and (c) the message CAUTION POSSIBLE INTERMEDIATE RESULT OVERFLOW is printed. The number of integer decimal places carried in such cases is given in the next sections.

Types of computation

There are two types of computation: mixed and unmixed. Mixed computations are those in which at least one operand is COMP-1 and at least one operand is not COMP-1. Unmixed computations are those in which the operands are either all

COMP-1 or all not COMP-1. Integer literals may be considered as either COMP-1 operands or decimal operands depending on the class of the other operands. Computations, mixed or unmixed, involving sterling fields are discussed separately on page 9.

LIMITS

1 The maximum permitted size of decimal operands is 18 digits including Ps. The maximum permitted size of sterling operands is 12 digits including Ps. The maximum permitted size of COMP-1 operands is 60 arithmetic bits including Ps. If these limits are exceeded the message ERROR : ILLEGAL OPERAND is printed and the statement ignored unless it is an ADD/SUBTRACT with multiple operands, in which case the offending operand is ignored.

2 COMPUTE statements and IF and PERFORM statements with arithmetic expressions must be unmixed. If not, the statement is ignored and the message ERROR : ILLEGAL OPERAND is printed.

3 In exponentiation, the only legal exponents are .5, positive integral literals and identifiers. Zero is permitted and produces an answer of 1. If an illegal exponent is found the same action is taken as for other illegal operands.

The contents of data-names representing non-integral numbers are truncated until integral, which may produce a zero field, and negative data exponents are made positive.

If the argument is negative the result of the exponentiation has the correct sign unless the exponent is .5 in which case the result is positive.

If the exponent is greater than one word only the least significant word is used.

UNMIXED COMPUTATIONS

With the following definitions the table below gives the number of integer and decimal places, that is, places after the point, carried both in intermediate results and in final results.

LIMIT	is 30 digits if the operands are decimal 100 bits if the operands are COMP-1
DMAX	is the maximum number of decimal places in any operand, including the receiving field
OP1	is the first operand
OP2	is the second operand
I1,I2	are the numbers of integer places in OP1 and OP2
D1,D2	are the numbers of decimal places in OP1 and OP2
DR	is the number of integer places in the receiving field (+1 if rounding is specified)

Operation	*Number of decimal places carried*	*Number of integer places carried*	*Number of integer and decimal places carried if LIMIT is exceeded*			
			COMPUTE and IF statements		*Other verbs*	
			Decimal	*Integer*	*Decimal*	*Integer*
Addition or subtraction	Max (D1, D2)	Max (I1, I2) +1	DMAX	LIMIT-DMAX	DMAX	LIMIT-DMAX
Multiplication	D1 + D2	I1 + I2	DMAX	LIMIT-DMAX	DR	LIMIT-DR
Division	Max ((D1-D2),DMAX) Note: D1 is the dividend	I1 + D2	DMAX	LIMIT-DMAX	DR	LIMIT-DR
Exponentiation	If the exponent is an integral literal, Min(D1 *OP2, DMAX) If D1=0,0 Else DMAX Note: OP2 is the exponent	If the exponent is .5 I1 Else I1*OP2	DMAX	LIMIT-DMAX	Not applicable	

ADD abd SUBTRACT statements

The composite is the sum of the moduli, or absolute values, of the maximum possible values of all the addends or subtrahends and must not exceed 18 decimal digits in length. If it does, the compiler will output the message COMPOSITE TOO BIG and will assume that the run time result of the computation will never be longer than 18 decimal digits. If this assumption is not true, the results of the computation will be unpredictable.

MULTIPLY statements

The composite is within range on decimal multiplication if:

(*the number of places before the point in the multiplicand*)
+ (*the number of places before the point in the multiplier*)
+ (*the number of places after the point in the receiving field*)
(+1, if ROUNDED is specified)

is less than 19.

If the composite is out of range, the compiler will output the message COMPOSITE TOO BIG and assume that the run time result will nevertheless satisfy the above conditions. If it does not, the results of the computation will be unpredictable.

DIVIDE statements

The composite is within range if:

(*the number of places before the point in the dividend*)
+ (*the number of places after the point in the divisor*)
+ (*the number of places after the point in the receiving field*)
(+1 if ROUNDED is specified)

is less than 19.

DIVIDE with REMAINDER statements

The remainder satisfies the equation:

*dividend = quotient * divisor + remainder*

and if rounding is specified, it is done after calculating the remainder.

If the composite is not within range, the compiler will print the message COMPOSITE TOO BIG. In this case the result will lose N decimal places off the least significant end, where N is the number of digits by which the composite exceeds 18, if ROUNDED was not specified, or (N-1) digits, if ROUNDED was specified. In the latter case, moreover, rounding will not take place.

MIXED COMPUTATIONS

A computation is considered as mixed if at least one of the operands is binary and at least one other is either a decimal field, a sterling field or a non-integral numeric literal.

Numeric MOVE

A MOVE of a binary to a decimal field or vice versa is considered an arithmetic operation since arithmetic conversion takes place. The decimal field may not exceed 18 decimal digits in length (12 digits if sterling).

1 BINARY TO DECIMAL The result of a MOVE is out of range on conversion if:

$DP + (BS - BP) \log_{10}2$ is greater than 18 (12 if sterling)

where DP is the point position of the decimal field
BS is the size in digits of the binary field
BP is the point position of the binary field

2 DECIMAL TO BINARY The result of a MOVE is out of range on conversion if:

$BP + (DS - DP) \log_{2}10$ is greater than 60

where BP is the point position of the binary field
DS is the size in digits of the decimal field
DP is the point position of the decimal field

Note: For the purposes of the calculations .3010 and 3.323 can be taken for the values of $\log_{10}2$ and $\log_{2}10$ respectively.

If the result of a mixed MOVE is out of range on conversion, the compiler assumes that the run time value of the source field is small enough for the result to be within range.

The value of the point position of a binary field in a mixed MOVE must be in the range -60 to +60. Similarly, the point position of a decimal field must be in the range -18 to +18. (-12 to +12 if sterling).

ADD and SUBTRACT statements

The COMP-1 and non-COMP-1 fields are added together to form two subtotals. Literals are added to the decimal sub-total. The rules for forming the subtotals are exactly the same as for unmixed cases. If all the result fields are COMP-1, the decimal subtotal is converted to binary with a point position = Max (BP, $DP\log_{2}10$)

where BP is the maximum point position of the COMP-1 field
DP is the point position of the non-COMP-1 subtotal

In all other cases the COMP-1 subtotal is converted to decimal with a point position = Max (DP, BP$\log_{10}2$)

where DP is the maximum point position of the non-COMP-1 fields
BP is the point position of the COMP-1 subtotal

The subtotals are then added together obeying the rules for unmixed cases.

Note: If there are mixed result fields the final decimal sum will be converted back to binary for storing in the COMP-1 fields.

MULTIPLY statements

The bit size of a decimal field is *digit size* *3.323 (3.323=$\log_2 10$)

Then if

bit size of multiplier + *bit size of multiplicand* -B -D $\log_2 10 > 10$

composite overflow occurs and accuracy may be lost on the left.

where B = *binary point position of multiplier + binary point position of multiplicant - binary point position of receiving field*

D = *decimal point position of multiplier + decimal point position of multiplicant - decimal point position of receiving field*

Note: The binary point position of a non-COMP-1 field is zero and the decimal point position of a COMP-1 field is zero.

DIVIDE statements

The bit size, N, of the quotient is defined as

bit size of dividend -B -D $\log_2 10$

where B = *binary point position of dividend - binary point position of divisor - binary point position of receiving field*

D = *decimal point position of dividend - decimal point position of divisor - decimal point position of receiving field*

If N > 100 composite overflow occurs and accuracy may be lost on the left.

If a remainder is specified, it is moved to the remainder field after calculation of the quotient.

In all mixed and unmixed computations 1 is added to the point position of the receiving field if rounding is specified.

STERLING FIELDS

The most important thing to appreciate about sterling fields in arithmetic operations is that the field is first converted to decimal pence and the resultant field operated on in exactly the same way as a decimal field. The rules for forming composites and detecting size error cases are exactly the same as those given in the earlier sections once the sterling field has been converted to pence. The maximum permitted size of a sterling operand is 12 digits.

If the sterling fields are integral or have point left, it is fairly obvious how to convert them to decimal: multiply the maximum contents of the pounds places by 240, the maximum contents of the shillings places by 12 and add the results to the maximum of the pence and decimal of pence places.

The less common point right cases are more difficult since the point positions of the equivalent decimal quantities are not generally the same as those of the original sterling quantities. For example, 99PP USAGE DISPLAY-1 represents a sterling field in the range £0 to £99. Given any value N in the field the equivalent number of pence is N × 240. Instead of holding this decimal equivalent as point 3 right the compiler holds it as point 1 right since the decimal equivalent of a point 3 right sterling field will always be a multiple of 10. If the decimal equivalent were represented as point 3 right, only multiples of 1000 could be held and a value such as £56 (decimal equivalent 13440) could not be held accurately in its decimal equivalent.

Again, for example, consider a field specified as 999PPP USAGE DISPLAY-2 which has two positions of pounds and one of tens of shillings. The maximum decimal equivalent is therefore (99 × 240) + (1 × 120), that is, 23880. It is clear that the decimal equivalent of any value contained in the field must be a multiple of 10. Because this is so, the compiler regards the maximum value as 2388 point 1 right and the user too should regard it as such when calculating composites and size error possibilities.

Therefore, when a point right sterling field is converted to its decimal equivalent, the point position of the field is very often changed too as can be seen from the following table:

Sterling point	*Point position of decimal equivalent*	
	If DISPLAY-1	*If DISPLAY-2*
1 Right	0	1 Right
2 Right	1 Right	0
3 Right	1 Right	1 Right
4 Right	2 Right	1 Right
.	.	.
.	.	.
.	.	.
(4+N) Right	(2+N) Right	(1+N) Right

where N is the difference between 4 and the sterling point number whose decimal equivalent point position is required.

As has been seen the total size of the equivalent maximum decimal quantity must not be more than 18 digits. The rules given above for the calculation of maximum equivalent decimal quantities are sufficient for the user to decide whether or not his sterling fields are legal arithmetic operands. However, since in practice working out the equivalent decimal could be rather cumbersome, the following simple general rule can be used for most cases:

A sterling field is a legal arithmetic operand if the number of places of pounds + the number of places of decimals of pence is less than 10.

Rounding with sterling receiving fields

If the receiving field has point left or is integral, the rounding rule is the same as for decimal computations, that is, 1 is added to the receiving fields after truncation if the most significant digit lost by truncation is greater than or equal to 5.

Example

```
ADD 3.4,A GIVING B ROUNDED
```

where A has PICTURE 9(3)V9 USAGE DISPLAY-1 and current contents 19;1 (that is, 19/11.1d) and B has PICTURE 9(4) USAGE DISPLAY-1.

This would give 1003 (£1 - 0 - 3) as the result in B. If the literal had been 3.3, the result would have been £1 - 0 - 2.

If the receiving field has point right, rounding is performed by adding one half of the radix of the most significant place lost by truncation to the result held as a decimal equivalent.

Example

```
ADD 6.9, A GIVING B ROUNDED
```

where A has PICTURE 9(3) V9 USAGE DISPLAY-1 and current contents 19;1 (that is, 19/11.1d) and B has PICTURE 9(3)P USAGE DISPLAY-1. This would give a result in B of 101 (£1 - 1 - 0.) If the literal had been 6.8, the result would have been £1 - 0 - 0.

If the decimal point or equivalent decimal point of the result as well as the sterling point of the reciving field is point right, a special case of composite error can arise which is flagged by the compiler with the message POSSIBLE ROUNDING OVERFLOW. This occurs if:

(*the size in digits of the composite*)
\- (*the decimal point of the composite*)
\+ (*the equivalent decimal point of the receiving field*)

is greater than 12.

© International Computers Limited, Reading, 1971