P800M PROGRAMMER'S GUIDE 2

Volume III : Software Processors

P800M PROGRAMMER'S GUIDE 2

Volume III: Software Processors

Assembly Language
Assembler
Linkage Editor
Overlay Linkage Editor
Line Editor
Debugging Package
ROM Image Generator
Utility Programs

A publication of

Philips Data Systems B.V.
SSS-DOC
Apeldoorn, The Netherlands

Pub. No. 5122 991 27392

August 1977

Printed in The Netherlands

This Volume provides a description of the Assembly Language, Disc Processors and Utility programs for the P800M series computers with disc, paper tape, magnetic tape and cassette tape peripherals. Information is also given on the Memory Management Unit MMU which allows to extend the memory handling over 32k up to 128k 16-bit words.
This manual has been written for experienced programmers.
Each item is described in a separate part to allow for easy reference.

Great care has been taken to ensure that the information contained in this manual is accurate and complete. However, should any errors or omissions be discovered, or should any user wish to make a suggestion for improving this handbook, he is invited to write his comments on the sheet provided at the end of the book and send it to:

SSS-DOC

at the address on the opposite page.

Table of Contents

Preface III

Glossary of terms IX

PART 1 ASSEMBLY LANGUAGE 1-1

Introduction 1-3
Syntax description 1-5

Chapter 1 Format of source statements 1-7
Label field 1-7
Operation field 1-9
Operand field 1-10
Comment field 1-13
Addressing modes 1-14

Chapter 2 Functional operation of instructions 1-17
Load and Store instructions 1-17
Arithmetic instructions 1-17
Logical instructions 1-17
Character handling instructions 1-17
Branch instructions 1-18
Shift instructions. 1-20
Control instructions 1-20
I/O instructions 1-21
External transfer instructions 1-21
Move table instructions 1-21

Chapter 3 Assembly directives. 1-23
Program framework 1-24
IDENT 1-25
END 1-26

Linkage control 1-27
ENTRY 1-28
EXTRN 1-29
COMN. 1-30

Assembly control 1-32
IFT 1-33
IFF 1-33
XIF 1-33
STAB 1-34

AORG 1-35
RORG 1-35

Value definition 1-36
DATA 1-36
EQU. 1-38

Area reservation 1-39
RES 1-39

Listing control 1-40
EJECT 1-40
NLIST 1-40
LIST. 1-40

Symbol generation 1-41
FORM. 1-41
XFORM 1-45
GEN. 1-46
List of predefined symbols 1-48

Chapter 4 Programming considerations 1-49
Stand Alone or Monitor controlled programming. 1-49
Interrupt system 1-49
System stack 1-50
User stack 1-50
I/O Processor 1-52
Memory Management Unit MMU 1-54
Layout of segment table word 1-55
Memory Protect 1-55
Trap action 1-56
Simulation routine 1-56
Adaptation of P855M software to P800M software 1-56
Use of the RTN instruction 1-57

PART 2 ASSEMBLER 2-1

Introduction 2-3

Chapter 1 Input 2-5
Source Modules 2-5
ASM control command. 2-5

Chapter 2 Processing. 2-7

Chapter 3 Output 2-9

Object modules 2-9
Assembly listing 2-9
Error representation on the listing 2-9
Symbol table 2-10

Chapter 4 Error Messages 2-11

PART 3 LINKAGE EDITOR 3-1

Introduction 3-3

Chapter 1 Input 3-5
Temporary object file 3-5
User library. 3-5
System library. 3-5
LKE Control command. 3-5

Chapter 2 Processing. 3-7

Chapter 3 Output 3-9
Load Module 3-9
Error messages 3-10
Listing and Map. 3-11

PART 4 OVERLAY LINKAGE EDITOR 4-1

Introduction 4-3

Chapter 1 Overlay Technique 4-5

Chapter 2 Programming Considerations 4-7

Chapter 3 Processing the input file 4-9
Segmented Programs 4-9
Processing 4-10
Processing of Commons 4-11
Load Module 4-11

Chapter 4 Output of Overlay Linkage Editor 4-13
Load Module 4-13
Map and Symbol Table. 4-14
Error Messages 4-16
Fatal errors. 4-16
Non-fatal errors 4-17

PART 5 LINE EDITOR 5-1

Introduction 5-3

Chapter 1 Processing 5-5
Control messages 5-6

Chapter 2 Error Messages 5-9

PART 6 DEBUGGING PACKAGE 6-1

Introduction 6-3

Chapter 1 Processing 6-5
Input/Output 6-6

Chapter 2 Commands 6-7
Parameter syntax 6-7

PART 7 ROM Image Generator 7-1

Chapter 1 General Principles 7-5

Chapter 2 Operation 7-9

Chapter 3 Control Commands 7-11

PART 8 UTILITY PROGRAMS 8-1

Chapter 1 Loaders 8-3

Chapter 2 Dump program 8-5

APPENDICES A-1

Appendix A File codes A-3

Appendix B Non-ASCII format on paper tape B-1

Appendix C Object code C-1

Appendix D ASCII code D-1

Absolute addressing — addressing specific locations in memory (see also relocatable addressing)

Assembler — a system program which translates programs written in Assembly Language into binary object code

Bootstrap — a program provided for initial loading of the system

Breakpoint — address at which execution of program stops to allow further debugging

Character — eight bits, representing an integer, letter or other data item

Cluster — a set of data in object code

Common

blank common — an area to which external references can be made from one or more modules

labeled common — a predefined external reference which can be used in several modules

Debugging Package — a processor which allows the programmer to insert breakpoints in a load module and call debugging functions before execution of a program

Directive — an instruction used for providing a framework for a program or for guiding the assembly process

Effective memory address — address in memory where the actual information can be found

Entry point — a label to which an external reference is made

External reference — a reference to an entry point in another program or module

File code — one or two hexadecimal digits associated with an I/O device

Identifier	a character or a combination of characters used to label an instruction or a value which is to be referred to by other instructions
Internal symbol	identifier in a module
IPL	Initial Program Loader. A program to load the monitor
Label	identifier of max. six characters long, the first always being a letter
Line Editor	a processor which handles the additions and deletions in source files or user data files
Linkage Editor	a processor used to link independent object modules before execution
Load Module	program output by the Linkage Editor containing no external references
Location counter	counter used to assign a relative or absolute address to program elements
MMU	Memory Management Unit
Mnemonic	abbreviation for an instruction, as used in the operation code field of a source statement, to indicate a machine instruction or directive
Module	a part of a program, enclosed by an IDENT and END directive, which can be treated independently of the rest of the program
Monitor	a system program which supervises the loading, processing and execution of user programs, starts and supervises the operation of processors and initialises I/O operations
Object code	program as translated by a language translator and suitable to be input to the Linkage Editor
Operand	an expression indicating the address, value or register to be operated upon by the machine instruction

Overlay Linkage Editor	a processor used to link independent object modules and produce a segmented program
Pass	one program run
Real Time Clock	a mechanism by means of which the amount of computer time allocated to a program is measured and a signal is given when that period of time has ended
Relocatable addressing	addressing in relation to the beginning of a program, not to specific locations in memory. The relocation of the addresses is then done by the machine
Source statement	one line in a source program
Symbol	an identifier, used as an address value in the operand field of other instructions

PART 1 ASSEMBLY LANGUAGE

This part contains a description of the Assembly Language. In this description it is made clear how a programmer can write his programs using the instructions of the P800M Instruction Set as well as the directives which guide the assembly process when the program is input to the Assembler.

Programs for the P800M computers are written in a symbolic language closely related to the machine code. Each statement (or line) of the program relates to a single machine instruction or to a data item to be taken into account by an instruction.

To write programs in the Assembly Language, the user should be familiar with the syntax of the instructions, which are divided in the following main groups:
Load and store instructions
Arithmetic instructions
Logical instructions
Character handling instructions
Branch instructions
Shift instructions
Control instructions
Input/Output instructions
External transfer instructions
Move table instructions
Floating Point Processor instructions.

Each module of a program consists of a number of characters grouped into lines and each statement in a module is made up of the following characters:
Letters: A to Z inclusive

Digits: 0 to 9 inclusive

Delimiters:

+	plus
—	minus
*	asterisk
=	equal
'	apostrophe
,	comma
	blank
/	slash
(	left parenthesis
)	right parenthesis
.	period
:	colon

Location counter

The Assembler maintains a location counter which is a software counter used to assign a relative or absolute memory address to program elements. The location counter starts with a relative value equal to zero, or it starts at an absolute address defined by the AORG directive, at the beginning of an assembly. The value of the counter is incremented by 2 or a multiple of 2 depending on the kind of instruction given.

The current value of the location counter is referred to by an * in the operand field (see below). In absolute program sections * has an absolute value. In that case the value is incremented in the normal way and the value may be changed by a RES or RORG directive.

The location counter may take neither a negative relative value nor an odd value.

Symbols

A symbol is a character or a string of characters used to represent addresses or values. Symbols may appear in the label field as well as in the operand field of a statement.

Their syntax is the same as for the label (see under label field). Some symbols are predefined and have a special meaning for the Assembler e.g. * indicates the current value of the location counter, P is the instruction counter etc.

The following symbols are used to define the syntax of the P852M Assembly Language.

< > to enclose syntactic items
| the vertical stroke has the meaning of or
::= is composed of
[ ] the syntactic items between these brackets may be omitted
[ ] select one of the items between these brackets
␣ space

The following list contains the definition of all items used.

<statement> ::= [<label>]␣<operation code>[<operand>][<comments>][*<comments>]

<label> ::= <identifier>

<operation code> ::= <mnemonic>[S|(<cnd>)|L][*]<directive>

<operand> ::= [+|−]<term>[+|−]<-term>[+|−]<term>

<comments> ::= <characters>|*<characters>

<identifier> ::= <letter>|<identifier><letter>|<identifier><digit>|<identifier>

<mnemonic> ::= <letters representing operation code>

<S> ::= <store indicator>

(<cnd>) ::= <numerical condition value>|<condition mnemonic>

<numerical condition value> ::= 0|1|2|3|4|5|6|7

<condition mnemonic> ::= Z|P|N|O|E|G|L|A|U|NA|NR|NZ|NP|NE|NG|NL|NN|

<L> ::= < load indicator>

* ::= indirection

<directive> ::= <IDENT, END etc.> see chapter on directives

<DATA defined hexa constant>	:: =	<see DATA directive>
<module name>	:: =	<symbol>
<symbol>	:: =	<characters representing address or value>
<predefined expression>	:: =	<max. of two defined symbols>
<entry point name>	:: =	<identifier within reference module>
<external>	:: =	<identifier defined in other module>
<common-field definition list>	:: =	<commen field definition>,<common field definition> .....
<common field definition>	:: =	<common field name>[<common field length>]
<common field name>	:: =	<identifier>
<common field length>	:: =	predefined (absolute) expression
<internal symbol list>	:: =	<internal symbol>,<internal symbol> .....
<internal symbol>	:: =	<identifier>
<field definition>	:: =	<field length definition>|␣=|:␣<field value definition>
<field length definition>	:: =	<number of bits>
< = field value definition>	:: =	<value to be placed in field>
<:field value definition>	:: =	<address of word>
<field number>	:: =	<decimal integer>
<term>	:: =	<constant>|<symbol>
<constant>	:: =	<decimal constant>|<hexadecimal constant>|<character constant>
<decimal constant>	:: =	<digit>|<integer>
<hexadecimal constant>	:: =	<hexadecimal integer>
<character constant>	:: =	<letter>|<digit>|<delimeter>
<letter>	:: =	A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>	:: =	0|1|2|3|4|5|6|7|8|9
<delimiter>	:: =	+|−|*|=|'|,|␣|/|(|)|.|:
<integer>	:: =	<number>

A source module consists of a sequence of statements. The Assembler interprets each line as it is presented.
Statements can be divided in the following fields:

- label field
- operation field
- operand field
- comments field

<statement> :: = [<label>]␣ <operation code> ␣[<operand>]␣[<comments>]␣[<sequence nr>]
*[<comments>]

Each field has to be separated from the other by one (or more) blank character(s). Blanks may not appear in the fields themselves except when specified in a character constant or in a comments field.
Instead of blanks a backslash may be used for separation (see 1-13). One or more blanks at the beginning of a statement indicate that there is no label field. If there are more than ten blanks after the operation field all following characters are considered to be belonging to the comments field.
An * (asterisk) at the beginning of a statement identifies that line as a comments line.

LABEL FIELD

<label> :: = <identifier>
<identifier> :: =
<letter> | <identifier> <letter> | <identifier> <digit> | <identifier>

Labels (or identifiers) in a module are used for reference purpose to other statements in a module.
The Assembler assigns, in most cases, to each label a word address value which is the numerical equivalent (absolute or relocatable) of the label.

The maximum number of characters in a label recognised by the Assembler is six. The first of those must always be a letter. A label, however, may contain more than six characters but the additional characters will not be taken into account. If the label has already been allocated to another statement an error message is output.
Period signs in a label are not significant, e.g.

PHILIPS

data systems

DATA _____ PAGE ___ OF ___

PROBLEM Example of programming paper PROGRAMMER _____

label (1–6)	operation (10–18)	operand (19 20–30)	comments (40–70)	identification (73–80)
MODIIF	END	*		
LDKL	ADD BWF+80	WORK ADDRESS		
LDKL	A 12,0			
MODIIF1	LDKL	A 12,BWF01,1		
ADR	A 1,1,A 12			
LDR	A 9,A 1,1			
LDKL	A 13,MOVE			

L.A.B.E.L. has the same meaning as LABEL

The value of a label is normally regarded as relocatable, except when:

- an absolute address is equated by an EQU directive
- the label appears in an absolute program section (defined by the AORG directive and which is not equated by an EQU directive to a label previously defined as relocatable).

OPERATION FIELD

<operation code> :: = <mnemonic>[S|(<cnd>)|L][*]<assembly directive>

where:

<mnemonic>

The operation field normally contains the mnemonic of a standard instruction. It is possible, however, to generate one's own instruction mnemonic by means of the FORM, XFORM and GEN directives.

S

Allowed after the mnemonic of certain register to register and memory reference instructions. It indicates that the result of the operation must be stored in a memory word (bit 15 of the instruction is set to 1). In fact, S has to be considered as a part of the instruction mnemonic.

e.g. C1R and C1RS instructions are to be considered as two different instructions.

NOTE: It is allowed to have the S preceded by a period sign though the Assembler does not take this sign into account.
e.g. AD.S␣ = ADS␣

(<cnd>):: = <numerical condition value>|<condition mnemonic>
<numerical condition value> :: = 0/1/ . . . . . /7
<condition
mnemonic> :: = Z|P|N|O|E|G|L|A|R|U|NA|NR|NZ|NP|NE|NG|NL|NN

This indicator specifies the condition under which a conditional branch instruction is to be performed. The table belows shows how in the Extended Assembler the conditional mnemonics and numerial condition values may be used.

COND. REG CONTENTS		
	GENERAL	ARITHM.
0	(0)	(Z) ZERO
1	(1)	(P) POS.
2	(2)	(N) NEG.
3	(3)	(O) OVERFL.
		NOT - CONDITION
≠0	(4)	(NZ) NOT ZERO
≠1	(5)	(NP) NOT POS.
≠2	(6)	(NN) NOT NEG.
≠3	(7)	UNCONDITIONAL

.L
Allowed after the instruction mnemonic of a constant instruction. It specifies that the operand is contained in 16 bits i.e. that the instruction must be assembled as a "long" instruction.

*
Indicates the indirect addressing mode in a register to register or a memory reference instruction.

OPERAND FIELD

The operand field may contain an address expression, a register expression or constants associated with the current machine instruction or assembly directive or a combination of those.
The structure and meaning of the operand depends on the type of instruction and directive and is explained below.
All operand expressions must be separated by a comma.

Expression

<expression> :: = [+ | −]<term>[[+ | −]<term>[[+ | −]<term>]]
<term> :: = <constant> | <symbol>

NOTE: * is considered to be a symbol.

An expression may not refer to more than 2 symbols and may not refer to more than one register name. In the latter case it may not contain any other term.

(<CND>)		
	COMPARE	I/O
	(E) EQUAL (G) GREATER (L) LESS –	(A) ACCEPTED (R) REFUSED – (U) UNKNOWN
	(NE) NOT EQUAL (NG) NOT GREATER (NL) NOT LESS	(NA) NOT ACCEPTED (NR) NOT REFUSED –

Address expression
The address specified in a memory reference instruction can be either absolute or relocatable.
An *absolute address* is the actual address in memory where the information the user needs can be found.
A *relocatable address* is relative to the beginning of the program in which it appears.

The address expression may contain any of the following terms or a combination of them:

*	asterisk, which is a predefined expression representing the current value of the location counter. This counter is incremented by two or a multiple of two depending on the length of the instruction.
symbol	used to refer to an instruction or data word with the same identifier in its label field. The Assembler will convert the symbol to a relative address.
displacement value	which can be attached to * or <symbol> to indicate a word not labeled by an identifier.

Predefined expression
A predefined expression is an expression consisting of not more than two symbols, each of which is defined i.e. has been assigned a value. Some symbols are implicitely predefined in the Assembler (see page 1-48).

An expression may contain only one external reference. The remainder, if any, of such an expression must have a predefined absolute value. The combination of an external reference and a predefined absolute value may only be used for specifying the value of a 16-bit field. The table below shows the result of a combination of positive and negative absolute or relocatable values:

1st term / 2nd term	+ R	− R	+ A	− A
+ R	E	A	R	R
− R	A	E	E	E
+ A	R	E	A	A
− A	R	E	A	A

where:

R = relocatable
A = absolute
E = erroneous

Register expression
Register expressions are regarded as predefined expressions and consist of one or two characters. The register expressions recognised by the Assembler are:

P	P-register or instruction counter
A1 . . . A14	Registers 1 to 14 (general purpose registers)
A15	Register 15 (stack pointer)

Constants
A variety of constant types may be specified in the operand of an instruction or directive.

<constant> :: = <decimal constant> | <hexadecimal constant> | <character constant>

Decimal constants
<decimal constant> :: = <digit> | <integer>

The decimal constant is a digit or integer contained in an 8-bit character or 16-bit word whose value may range from 0 to 32767.

Hexadecimal constants
<hexadecimal constant> :: = /<hexa integer>|X'<hexa integer>'

The hexadecimal constant is considered to be hexadecimal value or bit string in the range from 0 to /FFFF.

Character constants
<character constant> :: = '<character>[<character>]'

A character constant is composed of a character string enclosed in single quotation marks. The string is composed of the characters described in the character set on page 1-3.

A character constant can be used with a machine instruction only if the constant consists of either one character (short constant) or two characters (long constant). Longer strings can be specified in a DATA directive. A single quote mark (') used as a character is specified by two consecutive single quote marks.

COMMENT FIELD

Comments are only for the programmer's benefit. They are included in the assembly listing but not in the generated object program.
A line is considered to be a comment line when the first 10 characters of that line are blanks or when the line starts with an asterisk.

INPUT OF SOURCE STATEMENTS AND CORRECTIONS

The user may type in the statements and corrections from the operator's typewriter. He may do so by counting the number of characters to obtain a neat output on the listing device.
This procedure is rather cumbersome when many statements have to be typed in. An easier way of input from the typewriter is by typing a backlash between the various parts of the statement.
Example:

1st col		10th col		19th col		40th col
label	␣	opcode	␣	operand	␣	comments

may be typed as follows:

label\opcode\operand\comments

without having to count for the first column of each field.

Example:

1st col		10th col		19th col		40th col
label	␣	opcode	␣	operand	␣	comments

may be typed as follows:

label\opcode\operand\comments

without having to count for the first column of each field.

Example:

```
DATAF\LDK\A4,4
\ABL(7)\HALT
DEVUN\LDK\A4,5
\ABL(7)\HALT
ADDIT\LDK\A1,0\SET INDEX REGISTER FOR BUFFER.
\LDK\A3,/00FF\LOGICAL CONSTANT INTO A3
```

ADDRESSING MODES

In Volume II we see how addressing takes place from a hardware point of view. The condition an instruction must fulfil to meet the requirements of the Assembler is explained on the preceding pages. Specific examples, with source statements and explanation concerning the arithmetic instructions AD and ADR are given to show the operation within the CPU.

See for the hardware operation of those instruction Volume II. The order in which the examples are given is in accordance with the description on those pages.

Direct addressing

AD A1,LABEL	The contents of the memory location with symbolic address LABEL are added to the contents of register A1. The result is placed in A1.
ADS A1,LABEL	The contents of the memory location with address LABEL are added to the contents of register A1. The result is stored in LABEL.

Indexed addressing

AD A2,LABEL,A10	The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is placed in A2.

ADS A2,LABEL,A10

The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is stored in the address: LABEL + contents of A10.

Indirect addressing

AD* A2,LABEL

The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in A2.

ADS* A2,LABEL

The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in the contents of LABEL.

Index Indirect addressing

AD* A2,LABEL,A10

LABEL is added to the contents of register A10. The result points to an address whose contents are added to the contents of register A2. The result hereof is placed in register A2.

ADS* A2,LABEL,A10

LABEL is added to the contents of register A10. The result points to an address whose contents are added to the contents of register A2. The result hereof is placed in the address obtained of A10.

Register to Register operation

ADR A1,A2

The contents of A2 are added to the contents of A1. The result is placed in A1.

Register addressing

ADR* A1,A2

The contents of the address pointed to by A2 are added to the contents of register A1. The result is placed in A1.

ADRS A1,A2

The contents of the address pointed to by A2 are added to the contents of A1. The result is stored in the address pointed to by A2.

LOAD AND STORE INSTRUCTIONS

Load instructions

Before the programmer can perform an operation on the contents of a memory location or a register its contents must be placed in one of the registers A1 through A15, an operation which is performed by the load instructions. The contents of any memory location or any register are loaded into any register or memory location where the operation will take place.
The contents of a number of memory locations may be loaded in the same number of consecutive registers by means of a multiple load instruction. The first register to be loaded always is register A1.
When working in system mode on a P857M with MMU board (see also page 1-54) also locations beyond 32k can be loaded as their addressing is taken care of by the MMU.

Store instructions

Companion to the load instructions mentioned above are the store instructions which store the contents of a register, or a number of consecutive registers, containing the result of an operation, into a register or into a memory location or in a number of memory locations.

ARITHMETIC INSTRUCTIONS

Arithmetic instructions perform the normal arithmetic functions such as add, subtract, multiply and divide. The instruction operand operates upon the contents of the specified instruction.
This type of instruction includes also the double add and double subtract instructions where operations take place on the contents of two consecutive memory addresses and registers A1 and A2.

LOGICAL INSTRUCTIONS

Instructions described under this heading are called logical instructions because they operate on binary information according to the rules of logic. The first operand which may be a memory location, a register (R1 or R3) or a constant is compared with the second operand, register R2. The result is placed in a register or possibly in memory. In the instruction set each logical instruction is given a description in which way the contents of a memory locations is ANDed or ORed.

CHARACTER HANDLING INSTRUCTIONS

Character handling instructions operate on a character level. Characters may be exchanged, compared or 8 bits of a constant may be placed in 8 bits of a register.

BRANCH INSTRUCTIONS

These instructions cause a branch to an address in memory either when a certain condition is fulfilled or unconditionally.
In branch instructions on condition the instruction mnemonic is followed by a number ranging from 1 thru 6, enclosed in brackets. When the number is (7) or omitted, the branch is unconditionally.
These numbers are compared with the contents of the condition register set by the previous instruction.
The condition number has the following meanings:

(0) branch if CR	= 0	(4) branch if CR	≠ 0
(1)	= 1	(5)	≠ 1
(2)	= 2	(6)	≠ 2
(3)	= 3	(7) unconditional branch	

Example:

```
          —
          —
          LDK     A2,4
LABEL     SUK     A2,1
          RB(4)   LABEL
          —
          —
```

The Extended Assembler allows to use, instead of a number, a condition mnemonic e.g. Z, E, A (see page 1-10).
Unconditional branches are made by the following instructions:

- absolute branch instruction or relative branch instruction without a condition indicator or when (7) is specified.
- CF, RTN, EX instructions.

Long format absolute instructions permit to branch, forward as well as backwards, to any address in the program. Short format absolute branch instruction may only branch to locations /0000 to /00FE. Relative forward and relative backward instructions may not skip backwards more than 127 locations and 128 locations forward.
The Assembler gives an error indication if the permissible branch range is exceeded.
The address to which control is to pass may be indicated in various ways:

1 By means of a symbolic address expression
ABL(3) LABEL

2 By an absolute address held in a register

```
ABR(7)  A5
```

3 By using a constant to indicate an absolute memory address (short constant)

```
AB  /84
```

4 By means of a displacement value added to or subtracted from the instruction counter value (RB and RF instructions only). This displacement value is computed by the Assembler from an address expression used in the operand and may not exceed more than 128 words forward or 127 backwards.

```
RB(0)  ZERO
```

Another group of branch instructions are the Call Function and Return from Function instructions. The Call Function instruction provides a link to a subroutine by branching to the first instruction of the subroutine. To be able to resume the execution of the main program after the subroutine has been executed the contents of the P-register and the Program Status Word are stored in the stack. When the last instruction of the subroutine (RTN) is executed the contents of P and PSW are restored.

A special group within the branch instructions is formed by the instructions EX, EXK and EXR.
These instructions allow to address a memory location of which the contents is the binary representation of another instruction. The latter instruction is executed before the program continues with the next instruction in sequence.

Example:

```
        —
        LDKL    A3,CIO
        LDKL    A4,SST
        —
        —
CIO     CIO     A,1,1,TY
        EXR*    A4          EXECUTE SST
        RB(4)   *-2
        —
        EXR*    A3          EXECUTE CIO
        —
SST     SST     A7,TY
        RB(4)   *-2
```

The Execute instruction may not refer to other EX, EXK or EXR instruction, or to Call Function, RTN or double format instructions.

SHIFT INSTRUCTIONS

Shift instructions operate on a bit level. These instructions allow to rotate the contents of one of the registers A1 thru A7 n positions in the direction and manner specified in the instruction. Double shift instructions permit to operate on two registers.

CONTROL INSTRUCTIONS

These instructions perform the control of the program by allowing the program to be interrupted or not or to reset an internal interrupt. Except for the LKM instruction, control instructions should only be used in Stand Alone programming.

INH and ENB are two companion instructions. The program part between these instructions is not interrupted as INH inhibits all interrupts. ENB sets the machine status to permit interrupt.

Example:

```
        IDENT   TEST
OUT     EQU     *
        RORG    OUT+/600
START   HLT
        INH
        LDK     A5,0
        LDKL    A11,BUF
        LDK     A2,0
AGAIN   CIO     A2,1,/30        } program inhibited
        RB(NA)  AGAIN
        LC      A3,BUFPT,A5
        —
        —
        ENB
```

The RIT instruction is used to reset an internal interrupt which was previously set by an interrupt from the control panel, power failure/automatic restart, real-time clock or by a program error.
The programmer may specified a 5-bit hexadecimal value in the operand of this instruction to clear specifiec interrupts.

```
INTRTC  RIT   /1B  Reset the real-time clock interrupt
```

I/O INSTRUCTIONS

I/O instructions handle the data transfer between the CPU and peripherals, the operation of control units for these peripherals and status control.
In monitor controlled programs the I/O functions, initiated by these instructions, are taken over by a general I/O routine which is called each time a LKM instruction followed by a DATA directive is used.
The user need therefore not to write his own I/O routines. When the programmer is to write a Stand Alone program he has to write his own I/O routines. Since there is no memory protection option, except when working with the Memory Management Unit MMU, the programmer must be careful not to overwrite parts of a program already in memory. Two instructions, RER and WER, may be used to address an external register. The function of these instructions is described on page 1-52.

EXTERNAL TRANSFER INSTRUCTIONS

These instructions may only be used in system mode. The instructions RER and WER may be used to address an external register. The function of these instructions is described on page 1-53.
The remainder of the instructions in this group are instructions involving the operation of the MMU in the P857M.
They permit to load the 16 registers on the MMU board with information pertaining to the max. 16 pages into which a program can be divided.

Example:

```
SEGTAB  DATA    /0000
        DATA    /0400
        —
        —
        DATA    /3000
        —
        —
        TL      SEGTAB
```

A table store instruction writes the contents of these registers on the MMU, which are updated during the program execution, back to the specified reserved locations.

MOVE TABLE INSTRUCTIONS

The instructions under this heading are only accepted on the P857M.
They allow to copy a string of consecutive memory locations into another area, or when working in system mode with MMU, a string of consecutive memory locations from a user area to a system area and vice versa.

Directives are used to provide a framework for a program and to guide the assembly process. The directives are written in the program and are printed on the assembly listing if the listing option is specified in the ASM CCI command (see page 2-5).
The table below gives a survey of which directives are accepted by the Assembler.

Directive	Meaning	page
IDENT	Program identification	1-25
END	End of assembly	1-26
ENTRY	Define entry point name	1-28
EXTRN	Define external references	1-29
COMN	Define common blocks	1-30
STAB	Define internal symbol table	1-34
AORG*	Assign absolute origin	1-35
RORG	Assign relative origin	1-35
IFF	If false	1-33
IFT	If true	1-33
XIF	End of condition	1-33
DATA	Data generation	1-36
EQU	Equate symbol to value	1-38
RES	Reserve memory area	1-39
EJECT	Continue listing on new page	1-40
LIST	Resume listing output	1-40
NLIST	Suspend listing output	1-40
FORM	Format definition	1-41
XFORM	Extension of FORM directive	1-45
GEN	Generation directive	1-46

* The user should be aware of the fact that the Disc Assembler accepts absolute addresses, but that the Disc Linkage Editor does not and will output an error message ABS.ADR.

The directives can be divided in the following groups according to their function:

- Program framework : IDENT, END
- Linkage control : ENTRY, EXTRN, COMN
- Assembly control : IFT, IFF, XIF, STAB, AORG, RORG
- Value definition : EQU, DATA
- Area reservation : RES
- Listing control : NLIST, LIST, EJECT
- Symbol generation : FORM, XFORM, GEN

PROGRAM FRAMEWORK

The directives IDENT and END form respectively the first and last statements in the module. They are mandatory. The module punched on tape must be followed by :EOS or :EOF.
The IDENT directive is used for identification purposes and the END directive generates the END cluster after which the assembly process is stopped and a symbol table is printed.

The IDENT directive specifies the name to be given to the object module output by the Assembler. It is used for identification purposes in selective loading or updating (see parts on Linkage Editor and Line Editor). This directive must always be present and must be the first statement in a source module.

Syntax

␣IDENT␣ <module name>

where:

<module name>	A symbol which is specified according to the rules for a label.

END of assembly

END

This directive must be the last statement in a module and terminates the assembly process by punching an :EOS mark.

Syntax

[< label >]␣END␣[< predefined expression >][, < symbol >]

where:

< label >	The label is given a relative value equal to the length of the relative section of the generated object program. This length includes the length of the optional symbol table (see STAB directive, page 1-34). The value is 0 if this module is absolute.
< predefined expression >	This expression, if present, gives the address of the first instruction to be performed in the program after loading.
< symbol >	This parameter gives an entry point name to the internal symbol table of the generated object program when the STAB directive has been assembled. The internal symbol table consists of a list of all relocatable symbols defined and their numerical equivalents.

LINKAGE CONTROL

Some modules which have to be grouped into one larger program contain references to identifiers defined in other modules.
By means of the directives ENTRY and EXTRN the user is able to refer to certain parts in other modules whereas the directive COMN allows to transfer data among several modules either written in Assembly Language or in FORTRAN.
By using a COMN the programmer can define one or more common blocks. Each common block may be divided in a number of subfields of varying length, each having a symbolic name which can be referred to directly but only in the module in which they are declared.
COMN blocks may be labeled or blank; a COMN block is labeled if a name is attached to it.
The Linkage Editor allocates a space to the blank common block at the end of the link-load or link-edit run (see Linkage Editor). This block is placed at the end of the entire program.
Labeled commons are placed at the end of the first module that refers to it.

The ENTRY, EXTRN and COMN directives must always follow immediately after the IDENT directive and in this order, though it is not necessary that the ENTRY as well as EXTRN and COMN are specified.

So: IDENT, ENTRY, EXTRN, COMN or
IDENT, EXTRN, COMN or
IDENT, ENTRY, COMN etc.

The ENTRY directive is used to declare entry points, i. e. labels which are defined in the current module and used as operands of another module.
This directive, if present, must follow the directive IDENT.

Syntax

␣ENTRY␣<entry point name>[,<entry point name>, . . ,<entry point name>]

where:

<entry point name> Can be referred to by an operand of an instruction in another module. The maximum number of entry points which can be specified in one ENTRY directive is determined by the length of one line.

Example (see also EXTRN)

```
        IDENT   PROG
        ENTRY   NUMB1, NUMB2, NUMB 3
        —
        —
        —
NUMB1   LDKL    A3, LABEL
        —
        —
NUMB2   ST      A6, REFER
        —
        —
NUMB3   CF      A14, EOS
        —
        —
        —
        END     START
```

The EXTRN directive is used to declare externals i.e. operands which are used in the current module and defined as labels in another module.
The directive must follow ENTRY, or IDENT when the directive ENTRY is not present.

Syntax

␣EXTRN␣ <external name>[,<external name> . . . ,<external name>]

where:

<external name> Name of external reference (label in other module). The maximum number of external names which can be specified in one EXTRN directive is determined by the length of one line.

Example (see also ENTRY)

```
        IDENT   ASMPRO
        EXTRN   NUMB2
        -
        -
        -
        CF      A14, NUMB2
        -
        -
        -
        END     START
```

The COMN directive facilitates communication between modules written in Assembly Language or FORTRAN. The directive is written as follows:

Syntax

[<label>]␣COMN␣ <common field definition list>

where:

<common field definition list> ::= <common field definition>[,<common field definition list>]

where:

<common field definition> ::= <common field name>[<common field length>]

where:

<common field name> ::= <identifier>
<common field length> ::= <predefined absolute expression>

If the parameter <common field length> is omitted the default value assumed by the Assembler is 1. The field length must be given in words.

Example
A␣COMN␣FVAL1 (3), FVAL2 (3), INTGV (10)

which defines a labeled common, named A, having the length

3 + 3 + 10 = 16 words.

A is defined as an external reference and common block name. Either the common block name itself or the subfield names may be referred to in the same module. The subfield names are then considered to be equivalent to:

<common block name> + <absolute displacement>

so,

LD␣A1, FVAL2 is equivalent to LD␣A1, A + 6

and

ST␣A2, INTGV + 18 is equivalent to ST␣A2, A + 30

The displacements in this example are counted in characters.
Blank commons can only be referred to by the subfield names defined in the operand field.

␣COMN␣VAL1 (3), VAL2 (4)
␣COMN␣VAL3 (9), VAL4 (10)

These directives define a blank common of 3 + 4 + 9 + 10 = 26 words.

VAL2, for instance, may be used in symbolic expressions and is equivalent to:

<blank common "name"> + 6

More than one blank common may be specified in one module.

ASSEMBLY CONTROL

When it is neccessary to check whether a certain condition is satisfied before assembling a number of source lines, the user may include the directives IFT, IFF and XIF. The assembly of the IDENT – END – XIF directives are never bypassed by IFT or IFF.

By means of the STAB directive the user may specify one or more internal symbols which are to be used for Debugging purposes. All these symbols must have been defined previously in the current module.
Common block names are handled as externals.

The RORG and AORG directives are used to reset the location counter to a relocatable or absolute value indicated in the operands of those two directives.

The AORG and RORG directives are respectively used to define an absolute module section and a relative module section. Although the AORG directive is available in the Disc Assembler, the Disc Linkage Editor will not accept absolute addresses. The AORG and RORG directives are only to be used for self-contained executable programs.

The RORG directive is used to reset the location counter to a relocatable value, indicated in the operand of this directive, after the AORG directive has set the location counter to give absolute addresses.

Those directives are only used in combination with the directive XIF to indicate that a block of instructions is to be assembled only if a certain condition is fulfilled. The assembly of the IDENT — END — XIF directives are never bypassed.

IFT (IF True)

The IFT directive specifies that the Assembler has to assemble the next source lines only if the condition stated by this directive is fulfilled.

Syntax
␣IFT␣ <predefined absolute expression> = <predefined absolute expression>

If the first parameter ≠ second parameter the source line(s) following IFT up to the next XIF directive are not assembled.

IFF (IF False)

Syntax
␣IFF␣ <predefined absolute expression> = <predefined absolute expression>

If the first parameter = the second parameter the source lines following IFF will not be assembled.

Syntax
␣XIF␣

This directive allows all subsequent statements to be assembled until a new IFT or IFF statement is encountered.

The STAB directive outputs at the end of the relocatable program section of the generated module one or several internal symbols to be used for Debugging purposes (internal symbol is the address given to a symbol in the program after assembly). All symbols must have been declared previously in the current module.
STAB must immediately precede the END directive.

Syntax

␣STAB␣ <internal symbol list>

where:

<internal symbol list> :: = <internal symbol>[,<internal symbol list>]

If the STAB directive does not contain a parameter in the operand field all internal symbols of the module will be included.
The programmer may not specify entry points, external reference names or commons. This directive is only taken into account when in the END directive the parameter <symbol> is specified which gives the name of the internal symbol table.

AORG — Assign absolute ORiGin — AORG

This directive assigns an even absolute value to the location counter. The location counter receives that value by specifying <predefined absolute expression>.
From the time AORG is given and until a RORG directive is given the location counter is incremented in the same way as if it were relative, i.e. by increments of 2 and 4 depending on the length of the instruction. All labels in an absolute module are given an absolute value unless they are equated to a predefined relative value by an EQU directive.
RB and RF instructions in an absolute program cannot refer to an address in a relocatable program section as the place from where this section will be loaded is not known.

Syntax

␣AORG␣ <predefined absolute expression>

RORG — assign Relative ORiGin — RORG

The RORG directive allows the user to specify the beginning of a relocatable module by assigning a relative value, which must always be even, to the location counter. Its value may never become negative. If RORG has no operand the location counter is given the last relocatable value it has previously received. This value is equal to the length of the relocatable module at the time this directive is assembled.

Syntax

␣RORG␣ [<predefined relocatable expression>]

VALUE DEFINITION

The directives DATA and EQU are used to define certain values in a module.

DATA **DATA generation** **DATA**

The DATA directive is used to assign a value to one or more words in the module, for inclusion in the object module.

Syntax

[<label>]␣DATA␣<data expression>

where:

<data expression> :: = [<expression>|'<character string>'][<data expression>]

<label> refers to a symbol in the operand field elsewhere in the module.

<data expression> the data expression may be:

- a decimal or hexadecimal constant
- an address expression
- a character string consisting of one to thirty-two ASCII characters enclosed by single quote marks. A series of words is generated, of two characters each, which are left justified. When the number of characters is odd the rightmost character of the last word is a space.

Example

The expression may contain a number of parameters which, in total, may generate no more than 16 words in memory.

␣DATA␣'ABC',/0A0D, 1,/A, 2,'DEF'

will generate the following words:

4142	'AB
4320	C␣
0A0D	/0A0D
0001	1
000A	/A
0002	2
4445	'DE
4620	F'␣

Example
When the user wishes to make an ECB he may do so as follows:

ECB␣DATA␣1, BUF2, 6, 0, 0, 0,

Example
DATA␣−0128, + 12,/3AB,−/A, LABEL, 'TEXT:'

will generate the following:

FF80	-128
000C	+ 12
03AB	/3AB
FFF6	−/A
<value>	LABEL
5445	'TE
5954	XT'
3A20	:␣

Identifiers are normally defined by being assigned memory values as they appear in the label field of an instruction. The EQU directive may be used to define an indentifier in a direct manner by assigning to it the value of an expression in the operand field. The symbol in the label field is made equivalent to the value in that operand field. This value may be absolute or relocatable.
A symbol, provided it differs from standard mnemonics and FORM-defined mnemonics, may be used as an operation mnemonic but may not be followed by an operand. The Assembler generates one code word each time this mnemonic appears in the operand field.

Syntax

<label> ␣ EQU ␣ <predefined expression>

Example

```
CT␣ EQU␣/41C4
        —
        —
        CT
        LDKL␣A1, CT
```

CT may now be used anywhere in the program to represent the value /41C4.

Example

```
VAL␣ EQU␣10
        —
        —
        —
        LDK ␣ A1, VAL
```

Example

```
LAB ␣ EQU ␣ *
```

LAB receives the value of the location counter. (equal to LAB RES 0)

Example

```
C:1␣EQU␣25
REG:3␣EQU␣A3
```

Each time the Assembler encounters C:1 or REG:3 they are replaced by 25 and A3 respectively.
LDK A1, C.1 → = LDK A1, 25
LDK REG: 3, 1 → = LDK A3, 1
LDK REG: 3, C.1 → = LDK A3, 25

AREA RESERVATION

The directive RES can be used to skip over an area in memory. The RES directive saves a memory area of a given length, specified in the operand, advancing the location counter by twice the number of words specified.

RES **REServe memory area** **RES**

The RES directive is used to reserve a number of memory words. The programmer may specify this number in the parameter. The location counter is incremented or decremented depending on the positive or negative value of that parameter. If positive, a memory area of the specified value is reserved. If negative, a memory area of the specified size before the place identified by <label>.
The value of the latter is not changed but the location counter is reset to a lower value by subtracting twice the value specified.

[<label>]␣RES␣<predefined absolute expression>

where:

<label>	receives the address of the first word of the reserved area.
<predefined absolute expression>	specifies the length of the area to be reserved.

If <predefined absolute expression> is 0 the location counter is not updated and, if <label> is specified, the statement is equivalent to

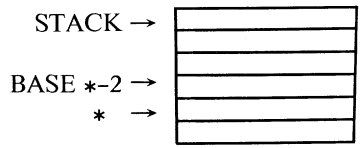
<label>␣EQU␣*

Examples:

```
        RES␣4      Reserve 4 words
LAB1␣   RES␣-2     Reserve 2 words before LAB1
INS␣    RES␣0      INS receives the value of the location counter.
```

Example of stack reservation.

```
STACK  RES 4
BASE   EQU *-2
```



LISTING CONTROL

The Assembler normally produces an output listing for each assembly. By means of the directives EJECT, NLIST and LIST the programmer may determine which parts of the modules do not need to be listed.

EJECT — Continue listing on new page — EJECT

This directive causes the remainder of the current page of the line printer paper to be left blank and the listing to be continued at the top of next page.

Syntax
␣EJECT␣

NLIST — Suspend listing — NLIST

The NLIST directive causes the Assembler listing to be suspended from the point where this directive is given until either the END directive or a LIST directive.
Lines which contain errors will continue to be printed during this phase.

Syntax
␣NLIST␣

LIST — Resume listing — LIST

The LIST directive causes the Assembler to resume the listing after it has been suspended by a NLIST directive.

Syntax
␣LIST␣

SYMBOL GENERATION

Three directives allow the user to make a number of special instructions for a specific purpose or program, namely FORM, XFORM and GEN. In the FORM directive the user may define the bit configuration and the mnemonic of the special instruction.
If two FORM-defined instructions are to be specified which differ only in the contents of certain fields the programmer may use the XFORM directive.
The GEN directive allows to include the instructions, defined by FORM and XFORM, in the existing Assembler by extending the Assembler's symbol table.
A particular useful pseudo-instruction or system macro can be defined once for all times instead of having to be generated by a FORM directive in every program where it is used.

FORM — FORMat definition — FORM

This directive is used to define the format of a word or a group of up to 8 words named by an identifier which can be used as an instruction mnemonic later in the program.
The directive is written as follows:

Syntax
< label > ␣FORM␣ < field definition >[, < field definition >, < field definition > ... < field definition >][/ < field number list >]

where:

< field definition > ::= < field length definition >[[= |:]field value definition >]
< field number list > ::= < field number >[, < field number list >]

and

< field number > ::= < decimal integer >

< **field length definition** > specifies the number of bits to be allocated to a field of the word and may range from 1 through 16. If several fields are defined inside a word the sum of the field lengths must be 16. The maximum number of consecutive words defined by a single FORM directive is 8.

<**field value definition**> can be used to place a value in the field to which it refers when the value is preceded by an equal sign (=).
If the value is preceded by a colon (:) the value indicates the address of a word in relation to the first word of the expansion defined by FORM. The value definition itself may be a predefined expression, an external reference without any displacement or a predefined absolute or relocatable expression. If a particular field has not received a value definition the field will be filled wit zeroes.

<**label**> defines the instruction mnemonic. The operand field of the directive must then contain values to be placed in any non-predefined fields. The last non-predefined value is default value.

Example
MNEM␣FORM␣16=/85A0,16:14,16=/8141,16=INST, 16, 16, 16

/85A0	→arithmetic or logical value
MNEM + 14	→address of word following this block
/8141	→arithmetic or logical value
INST	→identifier
0-0	3 words containing zeroes
0-0	
0-0	

The parameter 16:14 indicates a word address seven words from the beginning of the expansion defined by FORM. The programmer has to specify this address as the last three words are left zero.

Example
This example shows how the programmer may make an I/O request if not all parameters are known. By using the FORM directive he does not have to write the instruction sequence:

```
LDK    A7, –
LDKL   A8, DECB
LKM
DATA   1
```

```
00000                          IDENT  FORM
00001              INOUT       FORM   8=/07,8,16=/80A0,16,16=/2804,16=1
00002  0000        BUFFER      RES    10
00003  0014 0008   DECB        DATA   8,BUFFER,20,0,0,0
       0016 0000 R
       0018 0014
       001A 0000
       001C 0000
       001E 0000
00004  0020 0782   START       INOUT  /82,DECB
       0022 80A0
       0024 0014 R
       0026 2804
       0028 0001
00005  002A 2804                 LKM
00006  002C 0003                 DATA   3
00007                          END    START
       SYMBOL TABLE
BUFFER  0000 R   DECB   0014 R   START   0020   R
ASS.ERR.  00000
:EOF
```

From now on the programmer may use INOUT␣/82, DECB instead of LDK␣A7,...

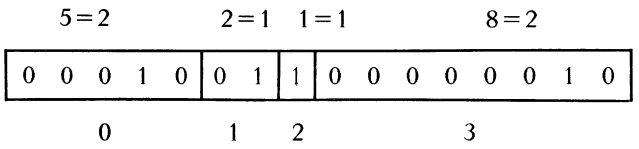
Field number list

If the programmer wishes to put the values of the operand field of the FORM defined mnemonic in an order different from that of the non-predefined field they are to occupy, or if the user wishes to alter the values held by any of the predefined fields, he must use the field number list parameter in the FORM directive.

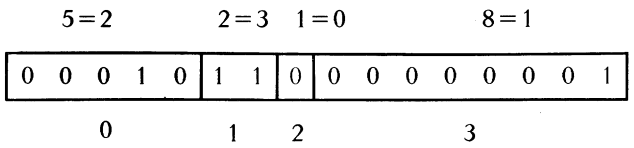
Each field that is generated is given a number, beginning with 0 for the first field, 1 for the second field, n-1 for the nth field (n may not exceed 15).

The field number list must be preceded by a / (slash) and be placed after the last field definition of the FORM directive.
All not predefined fields specified in the field definition list must also be specified in the field number list.
A field number is represented as a decimal integer.
If a field number list is specified after a FORM directive, the operand expressions following the pseudo-mnemonic will occupy the fields specified in the field number list in the given order. In this way, the contents of predefined fields may be altered while blank fields may be left blank.

Example
Suppose the user has specified in his program, by means of a FORM directive, a 16-bit word of the following format:



He wishes to have this word changed in:



He may do so by using the following instruction sequence in his program using the field number list in the FORM directive

```
        IDENT   EXAM
        —
        —
WORD    FORM    5=2,2=1,1=1,8=2/2,1,3
        —
        —
        WORD    0,3,1
        —
        —
        END
```

The Assembler will now change the fields as follows:
field no 2 (1=1) will be changed to contain the value 0
field no 1 (2=1) will be changed to contain the value 3
field no 3 (8=2) will be changed to contain the value 1
field no 0 (5=2) will keep the value 2.

The operand expressions following a pseudo-mnemonic are positional parameters. If one parameter is omitted (other than the rightmost one), its position must be indicated by a comma.

If a FORM defined mnemonic is identical with a standard instruction mnemonic, the pseudo-mnemonic is given priority.

Syntax

<label> ␣XFORM␣ <FORM-defined pseudo-mnemonic>,<field list>

The XFORM may be used each time two FORM-defined pseudo-mnemonics have to be defined which do not differ in the format but only in the values of the predefined fields.
The field list is a series of field definitions giving the format of the new pseudo-mnemonic and the contents of its fields.
The field length definitions must be the same as those of the FORM-directive referred to and appear in the same order.

Example

```
INST1␣FORM␣8=/FF, 4, 4, 16/1, 3, 2
INST2␣FORM␣8=/33,4,4,16/1,3,2
```

The XFORM directive combines the two and generates an INST2 instruction as follows:

```
INST2␣XFORM␣INST1,8=/33,4,4,16
```

The GEN directive allows to extend the Assembler symbol table so that it recognizes and assembles a number of non-standard symbols in any program in which they are used.

Syntax

␣GEN␣

Restrictions
The GEN directive may only be used in the source program in which it appears if it fulfills the following conditions:

- GEN must immediately precede END
- only the FORM, XFORM, EQU and EXTRN directives are allowed in this program

The Assembler does not verify if those conditions are fulfilled. It checks only if:

- object code is produced
- assembly errors have occurred

Example

```
        IDENT␣FORM
INOUT␣  FORM␣8 = /07,8,16 = /80A0,16,16 = /2804,16 = 1
        GEN
        END
```

The following procedure must be followed to include the features provided by GEN:

- link the Assembler with the generated object module by using the Linkage Editor.

This is done in the following way:

S: SCR /O	clear object file.
S: PLD ASM	Assembler is in load module format. It has to be extended with a new module. The Assembler is now output on punched tape in object code format.
S: RDO	Read object code and place it on /O file.
S: RDS	Read source program
S: ASM /S	
S: LKE N, M	
S: KPF /L, ASM	The new Assembler, with name ASM is placed in the library.

The output of this link-editing is the original Assembler extended with one new mnemonic.

NAME	MEANING	PREDEFINED VALUE	INTERNAL VALUE
P	Instruction Counter	0	0
A1	Register 1	1	2
A2	Register 2	2	4
A3	Register 3	3	6
A4	Register 4	4	8
A5	Register 5	5	10
A6	Register 6	6	12
A7	Register 7	7	14
A8	Register 8	8	1
A9	Register 9	9	3
A10	Register 10	10	5
A11	Register 11	11	7
A12	Register 12	12	9
A13	Register 13	13	11
A14	Register 14	14	13
A15	stack pointer	15	15

Note: P, A1, A2, A3 etc. can only be used to call the registers. If they are used for other purposes an error message will be output.

Data transfers between input/output devices and the central processor are controlled by device control units each of which may have one or several devices attached to it, depending on the type of device. Control units are attached to the central processor by an interrupt, or break line, by address lines and other signal lines which are used by the computer to determine whether a data transfer can be performed.
Data transfers take place through a channel, the General Purpose Bus. The actual programming of the data transfers may be on a character or word basis, where each word or character is programmed and transferred individually via the Programmed Channel or the user may program blocks of words or characters via the I/O Processor. In the latter case external registers may be addressed.

Stand Alone or Monitor controlled programming
The basic difference between Stand Alone programming and Monitor controlled programming is caused by the fact that in Stand Alone programming the user has to write his own input/output routines whereas in Monitor controlled programming the user may call certain monitor functions by means of *links to monitor* which execute the input/output.
For information on programming in either mode refer to the P800M Software Training Manual (Publication No 5122 991 1243X) and to page 1-55 of this manual.

Interrupt system
When working in interrupt mode each interrupt program may be connected to an interrupt level. As the actioning of an interrupt involves the direct accessing of the interrupt level's start address from its hardware interrupt location, the contents of this location must have been previously loaded with the correct address.
The start addresses loaded in these locations are not fixed and must be defined by the programmer.

interrupt level	*hardware interrupt location*
0 to 62	/0000 to /007C

where level 0 has the highest priority and 62 the lowest. The levels are defined at SYSGEN time (see Volume I).

System stack

To save the contents of registers when an interrupt is made into the main program, the hardware interrupt routine automatically uses register A15. This register addresses the stack which is to hold the contents of the P-register and the Program Status Word at the time the program was interrupted. It is therefore necessary to reserve sufficient space for the stack and to load register A15 with its start address. This may be done by using the appropriate assembly directives and by defining the start address by means of an identifier. The start address is the highest address reserved as the stack is filled from the high towards the lower addresses.

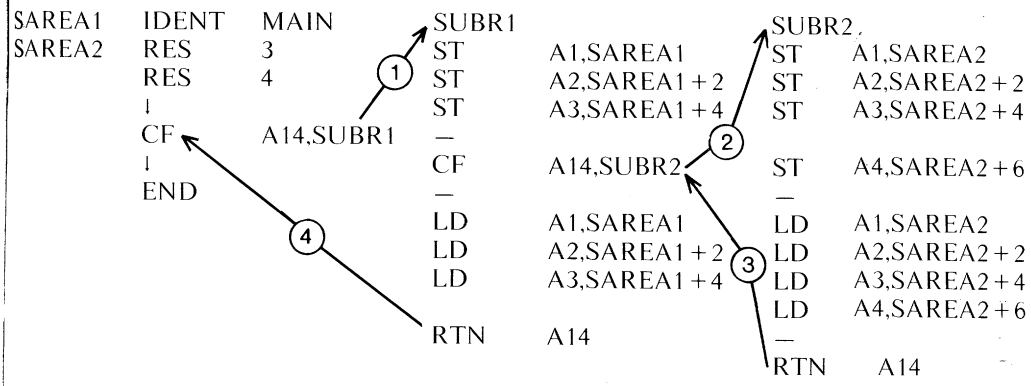
Apart from the contents of the P-register and PSW, the stack may be used to save the contents of other registers as required by the program. These registers are saved by means of Store instructions (1 for each register). Before returning to the main program, Load instructions are required to restore the contents of the stack, prior to RTN. During the hardware action further interrupts are inhibited. If the user wishes to allow the specific routine to be interrupted he must given an ENB instruction.

User stack

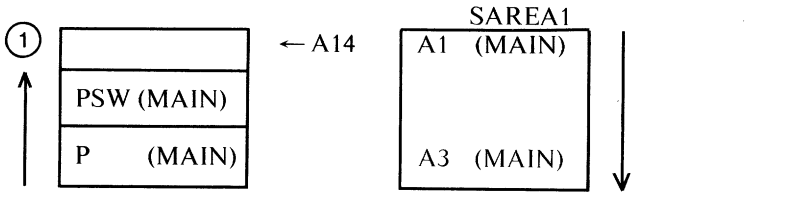
We have seen that with the A15 stack the P-register, the PSW and any other registers are saved with Store instructions in this stack towards the lower addresses. Now, if a user calls a subroutine with a CF instruction the contents of the P-register and the PSW are automatically stored in a stack he has set up previously, for example as follows:

```
        RES     20
STB     EQU     *-2
        LDKL    A 14,STB    then the subroutine is called:

        CF      A14,SUBR    and P and PSW are stored in the A 14 stack
                            (other registers may also be used as a stack pointer)
```

For example, for a program with two subroutines, one subroutine calling another one, the saving may be done as follows:

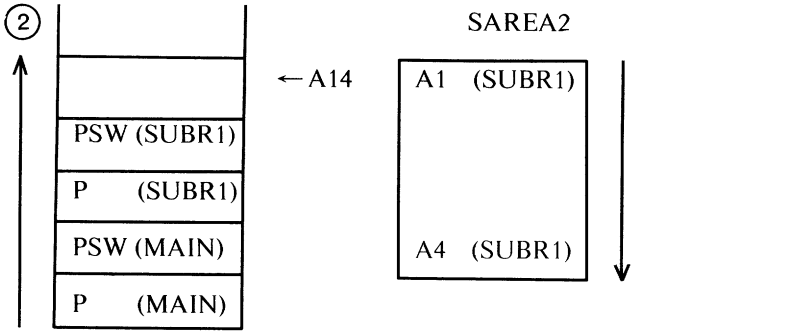


The following save operations take place in this example:



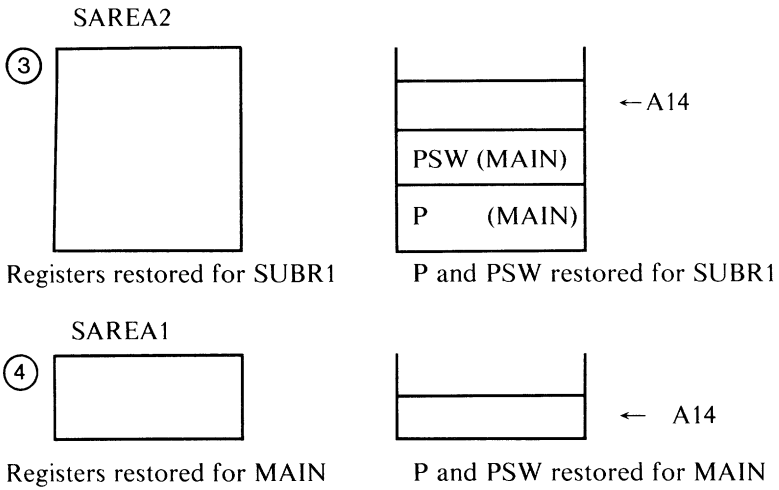
Stored automatically

Stored by user-written instruction



Stored automatically

Stored by user-written instruction



Note:
It is possible to return from SUBR2 directly to the main program but in such a case the user must update the A14 register contents i.e. the stackpointer, himself (with 4, in this case).

Memory Management Unit MMU
The MMU allows to extend memory addressing up to 128k words and can be used on the P857M.
Owing to this facility the P857M and its monitor are able to serve (a number of) large programs, each of which may not exceed 32k words. Programs of this size usually will be segmented, see Overlay Linkage Editor, and are stored on disc.
Apart from extended addressing the MMU also provides for memory protection.

Coding a program for operation with MMU and monitor requires no specific rules compared to e.g. the P856M as far as the memory addressing is concerned, as the addressing in an environment larger than 32k words, is transparent to the user.
Instructions relating to the MMU are only accepted in system mode.

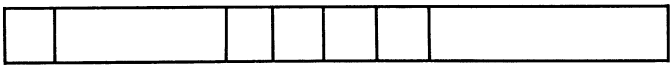
When the user program is called a path of n segments is loaded into memory, immediately after the monitor. These n segments are divided over parts of memory, called pages, of 2k words each. As more programs may be running simultaneously, the pages do not need to be loaded contiguously and may be dispersed over the entire memory available.
The monitor builds, for each program running, a table containing data where each page may be loaded and information particular to the page. This table is max. 16 words long and is loaded, by the monitor, in the 16 register segment table e.g. as follows:

```
LDR     A4,A11      where A11 contains the table address
—
—
TL      A4          where the table is loaded
```

To save the information in the MMU registers an ES or ESR instruction may be used (system mode only).

An address in the user program is divided in two parts. The four most significant bits point to a word in the segment table. The MMU translates these 4 bits into a 6-bit physical page address and takes the remainder of the instruction address as an address relative to the beginning of the page.

Layout of segment table word



bits 0 through 5 physical page address as derived from the four most significant bits in the instruction address

bit 6 Page error indicator. This bit is set by the monitor when a missing or wrong page is tried to be accessed The MMU will give a "Page Fault" interrupt. This bit is not used for system programs

bit 7 Read-only page. When this bit is set the page is protected against overwriting. A "Page Fault" interrupt is given when a program tries to write into it

bit 8 Modified page. This bit is set by the MMU when a write operation took place in this page. Instead of being overlaid the page is first written back onto disc before the area is used again

bit 9 Overflow. The setting of this bit depends on the value in bits 10 through 15

bits 10 through 15 Counter. A 6-bit counter is associated with each page descriptor. All counters are incremented at regular time intervals. This interval, which depends on the memory speed, is chosen at system generation time in a ratio 1 to 256. During execution of a program, each time a page of the running program is called the counter is reset to zero. If a counter reaches the interval set an overflow bit is set. When space in memory is required the Operating System swaps out those pages which have the overflow bit set.

Memory Protect

The memory protect facility of the MMU is obtained by setting bit 7 in the table

containing the words to be loaded in the MMU segment table registers. Remember, however, that instructions concerning the MMU are only accepted in system mode. If an attempt is made to access a protected page a "Page Fault" interrupt is given. This interrupt has the highest priority and causes storing in the system stack of:

- the address of the instruction which caused the interrupt
- the PSW
- a word containing the page address of the page in which the fault was detected, and the program level.

This interrupt is reset automatically after a branch has been made to the interrupt routine address.

Trap action

Instructions input to the P800M computer are checked and decoded by the CPU's hardware.
If an unexecutable instruction is encountered a *trap* action is started which consists of a hardware and software operation.
The hardware operation of the *trap* consists of the following actions:

- the cpu does not attempt to carry out the instruction
- interrupts are inhibited
- information which refers to the instruction's address and processor status (P and PSW) are saved
- an indirect branch is made to location /7E (start of trap routine)

The software operation of the *trap* consists of:

- save the address in P
- save the instruction's bit pattern and its second word, if any
- activate the Simulation routine (see below).

Simulation routine

The simulation routine allows the P852M user to simulate the following instructions:

multiply
divide
double add
double subtract
double shift
multiple load
multiple store.

This routine, which is activated each time an illegal instruction code is met in the instruction sequence, consists of two parts. One part analyzing the bit pattern saved by the trap routine and one part executing the instruction listed above.
The routine may be interrupted.

Adaptation of P855M software to P800M software

When P855M programs are to be adapted and run on the P800M computer the following points must be taken into account:

1 the sequence ... ENB INH ... in the P855M software permits to have the program interrupted after ENB to see whether an external interrupt is pending. As in the P800M external interrupts are not scanned at the end of a short instruction, a dummy instruction must be included after ENB to allow for an interrupt scan.
The sequence may be altered in ... ENB/RF* + 2/INH ...
2 in the P800M a stack overflow interrupt is given as long as the register A15 contents remain < /100. For the P855M a stack overflow interrupt is generated when the contents of register A15 = /100.

Use of the RTN instruction
Operation of the RTN instruction is slightly different for the P852M on one hand and the P856M and P857M on the other hand. The RTN instruction on the P852M reloads from the system or the user stack (the system stack is pointed to by register A15 and the user stack by one of the registers A1 through A14) the contents of the P register and the PSW as saved when the interrupt routine or subroutine was entered.
On the P856M and P857M the return is as follows:
When one of the registers A1 through A14 is specified, the P register and the CR field of the PSW in the user stack are reloaded. When register A15 is used as a stack pointer, the P register, bits 0 through 7, bit 9 and bit 15 are reloaded from the system stack.

Stand Alone Input and Output Programming

Programmed Channel

To control the data transfer between the device and the CPU the following instructions are, in general, available:

CIO Start	Start input or output
CIO Stop	Stop the input or output
INR	Input one character
OTR	Output one character
SST	Send status of the control unit
TST	Test if the control unit is busy

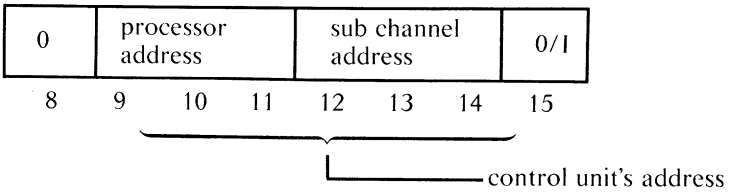
The register <r3> used in the CIO instruction must always contain additional information for the control unit e.g. input, output, parity, echo etc. Which information must be loaded can be found in the relevant hardware manuals delivered with the system.
When the CIO Start instruction is accepted (test the condition register) it is followed by an INR or OTR instruction. When the last character is transferred a CIO Stop

instruction must be given. This instruction should be followed by an SST instruction which gives the status of the relevant control unit and may reset an interrupt and switch a control unit to the Inactive State.

I/O Processor

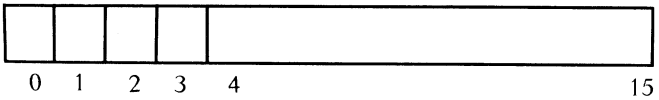
The I/O processor allows the high speed transfer of variable length or fixed length data blocks between a suitable control unit and the processor.
Up to eight I/O processors may be connected to the General Purpose Bus each of which may control up to eight control units via eight subchannels.
Each I/O processor has implemented two working registers which are used to effect register to register exchanges with the cpu internal registers.

Before a data transfer can be realised the user has to specify two control words for two external registers. These external registers are addressed by 2 WER instructions in which the address part must be composed as follows:



where processor and sub channel address are determined at system installation time. Both addresses, which may range from 0 thru 7, form together the attached control unit address. Bit 15 determines which control word is sent:
bit 15 = 0 1st control word
1 2nd control word

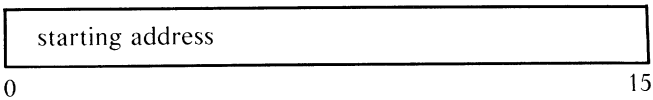
Format of control words
The format of the first control word is:



where:

bit 0 = 1 exchange is in word mode
0 exchange is in character mode
bit 1 = 1 exchange is from memory to control unit (output)
0 exchange is from control unit to memory (input)
bit 2 = 0
bit 3 = 0
bits 4 thru 15 specify the number of characters or words to be transferred.

The format of the second control word is:



When operating in *word mode* the 1st word of the block is always even (bit 15=0). In *character mode*, and bit 15=1, the right hand character is addressed (odd address), When bit 15=0 the left hand character is addressed (even address).

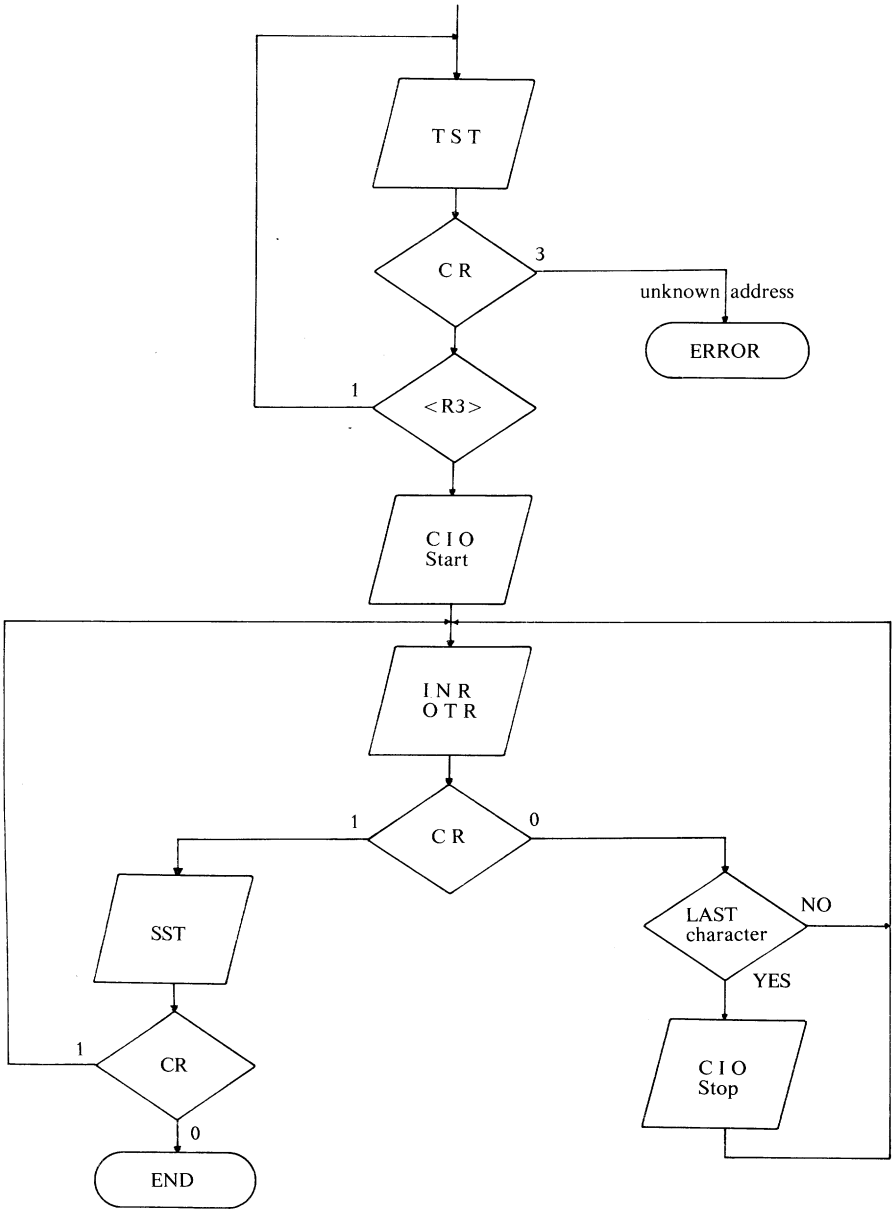
Example:

```
—
—
LDKL   A1,/8032      word mode, input, 50 words
LDKL   A2,BUF        starting address of block
WER    A1/A          send control words (0001010 and 0001011)
WER    A2,/B
—
—
CIO    A4,1,/01      start input (address: 000001)
—
—
```

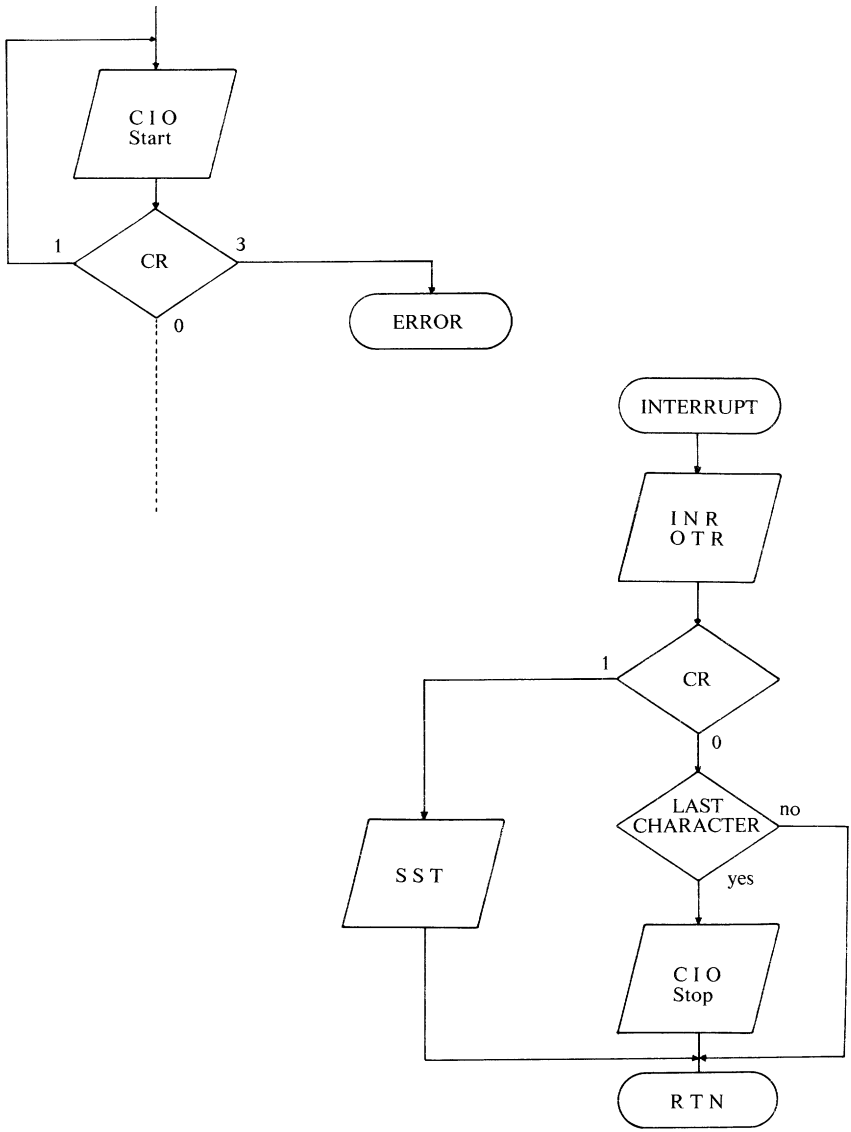
The RER instruction may now be used to read a transfer's effective length after termination of the I/O operation.

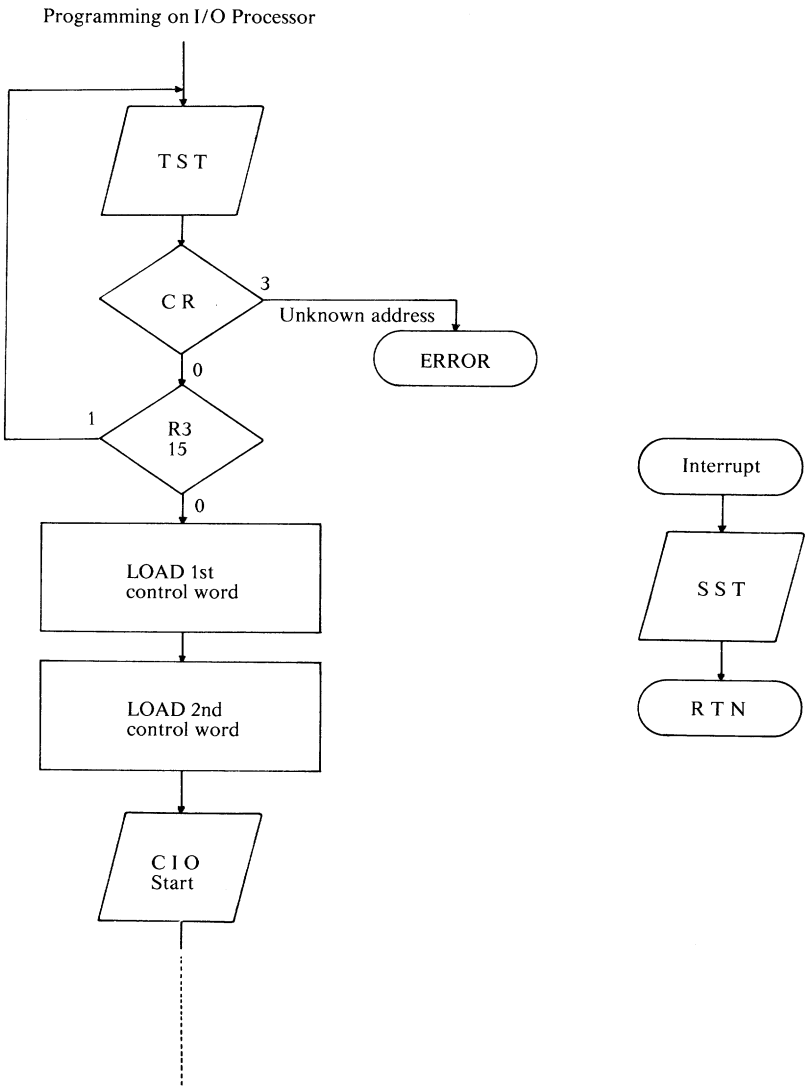
When the exchange is completed an SST instruction should check the status of the control unit and set it to the *Inactive* state. The control unit may now be re-initialised for a new transfer.

Input/Output Programming on Programmed Channel
a) without interrupts



b) with interrupt handling




Programming on I/O Processor
TST
CR
3
Unknown address
ERROR
0
1
R3
15
0
LOAD 1st
control word
LOAD 2nd
control word
CIO
Start
Interrupt
SST
RTN

```
00000                          IDENT    OUTPUT
00001                          *
00002                          * EXAMPLE OF STAND ALONE PROGRAM TO OUTPUT A
00003                          * MESSAGE ON THE TELETYPE LOG AND NEXT HAVE THE
00004                          * SAME MESSAGE PUNCHED ON THE TELETYPE PUNCH UNIT.
00005                          *
00006                          *
00007  0000  2044             MESSGE   DATA     ' DIT IS EEN TEST',/000A
       0002  4954
       0004  2049
       0006  5320
       0008  4545
       000A  4E20
       000C  5445
       000E  5354
       0010  0D0A
00008  0012  207F             START    HLT
00009  0014  20BF                      INH
00010  0016  0112                      LDK      A1,18                 COUNTER FOR NO OF CHARACTERS
00011  0018  80A0                      LDKL     A8,0
       001A  0000
00012  001C  0600                      LDK      A6,0
00013  001E  46D0                      CIO      A6,1,/10              START TELETYPE IN OUTPUT
00014  0020  5C04                      RB(NA)   *-2                   ACCEPTED?
00015  0022  E542             ASR      LC       A5,MESSGE,A8          LOAD A CHAR IN A5
       0024  0800  R
00016  0026  4510                      OTR      A5,0,/10
00017  0028  5C04                      RB(NA)   *-2                   ACCEPTED?
00018  002A  90A0                      ADKL     A8,1                  POINT TO NEXT CHARACTER
       002C  0001
00019  002E  1901                      SUK      A1,1
00020  0030  5C10                      RB(NZ)   ASR                   ALL CHARACTERS PRINTED?
00021  0032  4690                      CIO      A6,0,/10              YES, SWITCH TELETYPE OFF
00022  0034  5C04                      RB(NA)   *-2                   ACCEPTED?
00023  0036  4CD0                      SST      A4,/10                SEND STATUS
00024  0038  5C04                      RB(NA)   *-2                   ACCEPTED?
00025                          *
00026                          * PUNCH THE MESSAGE
00027                          *
00028  003A  80A0                      LDKL     A8,0
       003C  0000
00029  003E  0112                      LDK      A1,18                 COUNTER FOR NO OF CHARACTERS
00030  0040  0600                      LDK      A6,0
00031  0042  46D0                      CIO      A6,1,/10              SWITCH TELETYPE ON IN OUTPUT
00032  0044  5C04                      RB(NA)   *-2                   ACCEPTED?
00033  0046  0512                      LDK      A5,/12                * SWITCH PUNCH UNIT ON
00034  0048  4510                      OTR      A5,0,/10
00035  004A  5C04                      RB(NA)   *-2
00036  004C  E542             PTP      LC       A5,MESSGE,A8
       004E  0000  R
00037  0050  4510                      OTR      A5,0,/10              OUTPUT THE CHARACTER IN A5
00038  0052  5C04                      RB(NA)   *-2
00039  0054  90A0                      ADKL     A8,1
       0056  0001
00040  0058  1901                      SUK      A1,1                  ALL CHARACTERS PUNCHED?
00041  005A  5C10                      RB(NZ)   PTP
00042  005C  0514                      LDK      A5,/14                SWITCH PUNCH UNIT OFF
00043  005E  4510                      OTR      A5,0,/10
00044  0060  5C04                      RB(NA)   *-2                   ACCEPTED?
00045  0062  4690                      CIO      A6,0,/10
00046  0064  5C04                      RB(NA)   *-2
00047  0066  4CD0                      SST      A4,/10
00048  0068  5C04                      RB(NA)   *-2
00049  006A  207F                      HLT
00050                                  END      START
```

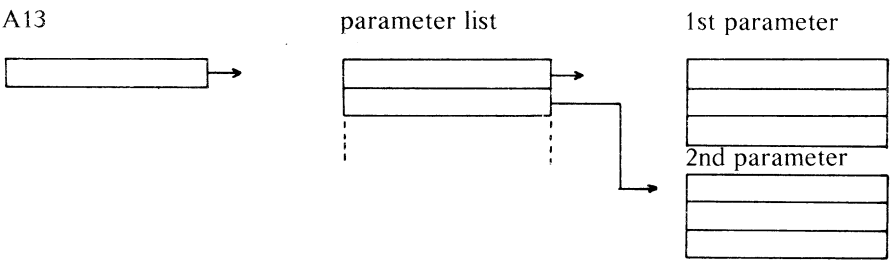
Source program calling a subroutine in FORTRAN library

When writing a program in Assembly Language it may be useful to have a certain operation performed by a subroutine which has been specifically included in the FORTRAN library to execute such a function.
The user may call this subroutine, in his Assembly program, in the following way:

Suppose the user wishes to multiply two floating point numbers. The FORTRAN library subroutine, which executes this multiplication, has F:RM as entry point. The framework of the Assembly program, with only the relevant details, is written as follows:

```
            IDENT   ASMPRO
            EXTRN   F:RM

FLNUM1      DATA    —
            DATA    —
            DATA    —

FLNUM2      DATA    —
            DATA    —
            DATA    —

            LDKL    A13, PARLIS
            CF      A14, F:RM

PARLIS      DATA    FLNUM1
            DATA    FLNUM2
```

Before the CF instruction is executed, register A13 must contain the address of a parameter list. This list must contain the address of floating point number 1 and the address of floating point number 2.



The subroutine in the library contains the following relevant items:

```
IDENT    FRTLIB
ENTRY    F:RM
—
—
—
—

RTN      A14
```

This subroutine does not use the stack of the calling program, except for the return. When values are to be returned to the main program an integer will be returned to A1 and a real value to the registers A1 to A3 inclusive (mantissa in A1, A2 and the exponent in A3).

The main program must now be link-edited or link-loaded with the called subroutine and the FORTRAN library.

The Linkage Editor selects those modules required for program execution.

PART 2 ASSEMBLER

The Disc Assembler is a one-pass processor which is written in the system library.
This Assembler operates under control of the Disc Operating Monitor (DOM) and translates the source programs, input via a source input unit, into object code.

The I/O requests from a program are handled by an Event Control Block. This Block contains, among other information, a file code specifying a physical unit or a specific file on disc.
A list of the file codes is given in Appendix A

Source modules
Source modules may be read from the source input device. Those modules are placed in the temporary /S file where they are kept until they are processed by the Assembler.
When more than one module is input the modules must be separated by an :EOS mark. The last module must be terminated by an :EOF mark.
When the file code specified in the ECB refers to a source file on disc this file may be found in:
— a library (on disc)
— a temporary file.

ASM control command
The ASM control command may be introduced after the user has identified himself by the user identification and calls the Assembler from the system library. In this command may be specified a number of (optional) parameters.
The syntax of this control command is:

ASM␣[/S|<name>][,NL]

where:

/S	the source program to be assembled must be read from the /S file.
<name>	indicates the name of a library source module or program to be assembled.
NL	if this parameter is specified the Assembler will produce no listing of the assembled program. If NL is omitted a listing is produced on the print unit. Error messages, however, will always be listed.

The Control Command Interpreter checks the ASM control command parameters for errors.
When there was an error in the parameters an error message indicating the error is printed, followed by the printing of S: at the beginning of the next line. The user may now input the correct ASM command:

```
S:ASM
  FILE NAME MISSING
S:PROG,NL
```

The source modules are now read and assembled.

Errors in the source program are detected by the Assembler. It is not possible, however, to correct errors during processing.
In case of a fatal error during processing (I/O error, Table Overflow etc.) the source input is read until an :EOF mark is encountered. At that moment the following message is printed:

FATAL ERROR HAS OCCURRED NO OBJECT CODE PRODUCED

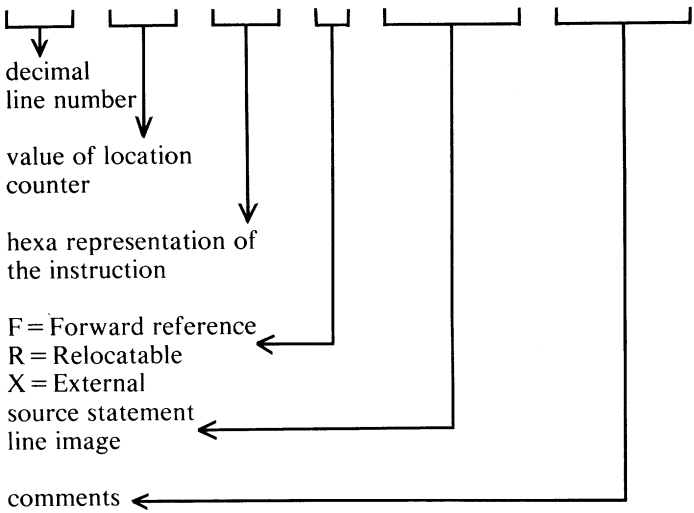
and the object file on which the output of the Assembler was written, is deleted.

Object modules

The standard object output file for the Assembler is the temporary /O file. If this file does not yet exist an assignment is made for it. When it did exist the object output is written after the information already existing in this file unless it has been closed by an EOF record. In that case a new /O file is created and the old one is deleted.

Assembly listing

The assembly listing is output on the listing device if the NL option is omitted. The format of the print-out is:



* the sequence number of cards is not printed on the listing

Error representation on the listing

When a non-fatal error has occurred during assembly the processing continues but the place where the error occurred is indicated by: *R, *L etc. preceding the line in which the error occurred. An * is printed underneath the place where the error was detected.

An error counter is updated every time an error occurs. The number of errors is given after the printing of the symbol table, by:

ASS. ERR. <5 decimal digits> (see example)

Symbol table

Each label which appeared in the label field of an instruction is given an address relative to the beginning of the program.
The symbol table consists of a list of labels for each external name defined in or referred to within a module.
This table is always printed out even when the user did not ask for a listing.
The symbol table has the following format:

```
                SYMBOL TABLE

M:A00   0000  R   M:B00   005C  R   MPYMOD          X
ADDMOD        X   DSUMOD        X   SYSAB           X
M:A01   0006  R   T:OPC   0136  R   M:A02   001B    R
M:A03   001E  R   T:SVR   0152  R   M:A04   002F    R
M:B01   008C  R   M:B00A  0074  R   M:B00B  0082    R
M:B01A  00A2  R   M:B02   00AA  R   M:B03   00CA    R
M:B04   00DC  R   M:B04A  00EA  R   M:B07   0100    R
ENDSVR  024E  R

                ASS.ERR.    00000
```

This table will only be included in the object output, for debugging purposes, when a symbol table name has been declared in the END directive.

The following error messages are output by the Assembler:

CODE	MEANING	DESCRIPTION
*I	Illegal identifier	The first character of a symbol must be a letter.
*C	Illegal constant	– "constant" overflow. – A constant with hexadecimal value must begin with X / and end with /. – A constant should not have been written here. – Hexadecimal constant written either X″ or /″.
*X	Illegal expression	– More than two symbols defined. – More than three terms in the expression. – An external reference and a forward reference have been specified in the same expression. – An external reference is preceded by a minus sign. – A plus or minus sign is not followed by a term. – A forward reference or external reference is specified in a requested predefined expression. – A register expression may not contain more than one term.
*R	Illegal relocation	– Either a predefined expression or predefined relocatable section has been input. – Too many relocatable symbols are added or subtracted from each other. – The expression is equal to the result of a subtraction of a relocatable part from an absolute part.

CODE	MEANING	DESCRIPTION
		— If an external reference is specified the displacement value must be absolute. — The instruction code operation defined by an EQU directive must be absolute.
*L	Illegal label	— The label has been defined previously as: — a symbol name — an external reference name — entry point name. — A label has been given where it was not allowed. — A label must be specified.
*P	Illegal parameter	— Too many parameters specified in the operand of an instruction, in the pseudo instruction or directive. — Not enough parameters specified in the operand of an instruction, defined pseudo instruction or directive. — A parameter in the STAB directive may not be an entry point name, a COMMON name or a forward reference. — The operand in a DATA directive may not give more than 16 code words. — ″ is not a character string. — Illegal use of a register name in a standard instruction operand.
*O	Overdisplacement	— Displacement value of parameter too large.
*E	Not an even address	— The specified start address is not even. — The specified AORG or RORG operand is not even.
*M	Unknown mnemonic	— Unknown mnemonic. — Unknown condition mnemonic.

CODE	MEANING	DESCRIPTION
*S	Illegal statement	— The ENTRY or EXTRN or COMN directive is no longer acceptable. — The directive does not need an operand. — The directive needs an operand. — Invalid character. — Invalid indirect addressing. — Invalid condition specification. — The label is not followed by an operation code. — '(' is not followed by')'. — The operand value of RES directive makes the instruction counter value negative. — GEN cannot produce any code as either any error occurred or the code word has already been produced.
*F	Illegal FORM or XFORM directive	— An XFORM declared symbol must be linked to a FORM defined pseudo whose name is the first parameter of the operand. — More than 16 fields specified. — Negative field length. — The length of this field cannot be contained in 16 bits. It is too long. — A displacement value is not allowed when the predefinition concerns an external reference name. — The : predefinition is only allowed for a 16-bit field. — Invalid predefined value of a field (overdisplacement or negative value for a less than 16-bit field). — The division of the current word of an XFORM declaration is not the same as the corresponding word of the linked FORM symbol.

CODE	MEANING	DESCRIPTION
		— The predefinition of the fields of the current word of an XFORM declaration are not the same as the corresponding word of the linked FORM symbol. — More than 8 words described by a FORM declaration. — More than the number of words described by the linked FORM symbol described by an XFORM declaration. — The field number specified in the syntax definition line is invalid. — Twice the same field specified in the syntax definition line.
******O	Core overflow	— Fatal error. Too many symbols or forward references used.
******E	End missing	— Fatal error. The END statement is missing.
******I	IDENT missing	— Fatal error. The IDENT statement is missing.

PART 3 LINKAGE EDITOR

The Disc Linkage Editor is a one-pass processor designed to run under the Disc Operating Monitor. The processor is written on disc. The task of the Linkage Editor is to make from the object output of the language translators (Assembler and FORTRAN) an object program which can be loaded into memory and in which the external references are matched. This output program is called Load Module and is executed by the CCI command RUN.
When the output of the language translators is to be input to the Disc Linkage Editor it must not contain any absolute address. An error message ABS.ADR. is given should an absolute address be encountered.

In the only control command for the Linkage Editor the user may specify a number of options.
For debugging purposes he may specify whether a part or the whole of the entry point symbol table is to be added to the Load Module.

The input to the Linkage Editor may come from the following files:
- temporary object file
- the user library
- the standard library.

Temporary object file
This file is the main input to the Linkage Editor. On this file are written the object modules read from object input, the output of the language processors and the modules from the libraries required for processing.

User library
On this file are stored all object modules the user decided to keep. This file is scanned only by the Linkage Editor when in the LKE control command, see below, the option U is specified or when N, S and U are not specified.

System library
This file keeps the system library and contains all the object modules belonging to processor libraries. This file is only scanned when the option S is specified or when N, S and U are not specified.

LKE Control Command
The LKE Control Command is a command handled by the CCI and calls the Linkage Editor from the library:

LKE␣[N|S|U][,M][,[DE|DS]][,/ <address>][,<start address>]

where:

N	The system or user library do not need to be scanned
S	Only the system library has to be scanned
U	Only the user library has to be scanned
	Note: When neither of those three parameters is specified the Linkage Editor scans both the system and user library, the user library first.
M	The listing of the map, which consists of a listing of the module names and their loading address, and an alphabetical list of all entry points and common blocks together with their value, must be printed.

Example:

LKE␣M	The map must be printed and both libraries scanned
LKE␣	No map, both libraries have to be scanned

DE — The whole symbol table with all entry points is to be added to the load module. It will be used for debugging purposes.

DS — A symbol table containing only the internal symbol entry point list is to be added to the load module.

/ <address> — Hexadecimal displacement value of the blank common from the beginning of the load module.

If bit 15 of <address> = 1 relocatable blank common
0 absolute blank common

This option may be used for communication between several load modules making use of the same blank common.

The load modules may, in that case, be called by the Load Monitor request (LKM DATA 9) and must have been defined previously in a SEG command.

The address of the blank common must be specified in such a way that it will not be destroyed when a load module is loaded.

The address must be defined at link-edit time.

Default: The blank common, if any, is placed at the end of the load module.

<start address> — Name of the start address defined as an entry in one of the modules in the /O file.

The name must be different from the option parameters used in the LKE command: N, S, U, M, DE, and DS.

Default: start address is the last start address encountered in the /O file, if any.

Before the LKE is started an EOF record is written on the /O file, if not already done.

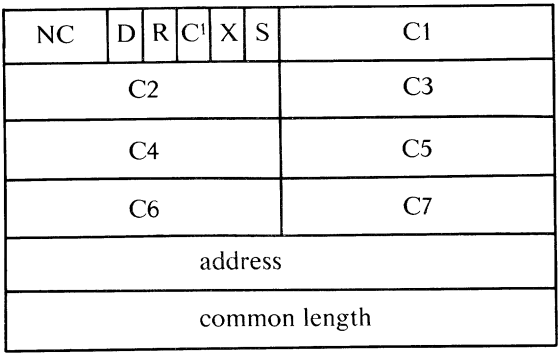
An assignment is made for the /L file onto which the LKE will write the Load Module. If a fatal error occurs during the link-edit operation the /L file is deleted and a RUN command, without parameters, is refused.

The Disc Linkage Editor processes the input object modules according to the cluster type encountered in the input stream (see Appendix C). When the LKE control command has been introduced the DLE starts linking together all object modules of the temporary object file by matching, as much as possible, the external references.
Once the temporary object file has been completely processed and there are still external references which could not be solved the very first time, the DLE will start scanning the current object libraries, either by default or by having the relevant parameter specified in the LKE command.

The user library is scanned first and the DLE looks for the requested entry points. If still not all external references are matched the DLE starts scanning the system library and link-edits all modules after having found the missing entry points. Should there still be left unsatisfied, the error message UNS.EXT. (unsatisfied external reference) is printed on the operator's typewriter, which they are will be listed in the map.

When one of the Debug options has been specified — DE or DS — the DLE adds to the generated load module a symbol table. When the DE option is given the symbol table contains all entry points of the link-edited modules (entry points of the internal symbol table are included).
When the DS option is chosen the symbol table contains only the entry points of the internal symbol table.

The symbol table is organised in the following way:



where:

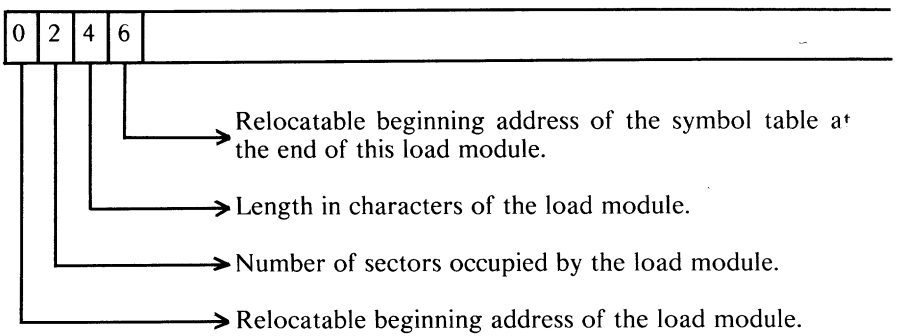
NC	Number of characters, which may vary from 1 to 7, of the symbol
D	= 1 if the symbol is defined = 0 if not
R	= 1 if the entry point is relocatable = 0 if absolute
C′	= 1 if the symbol is a common block name = 0 if not
X	= not used
S	= 1 if the symbol appears in an internal symbol table entry point name.
C	= Symbol. The number of characters in the symbol may vary from 1 to 7. They are coded in ASCII.
address	= Address of the entry point name or common block
common length	= Length of common.

The blank common (if any) is allocated at the end of the last link-edited module or it is allocated at the place specified in the LKE command by /<address>.
The processor checks that the blank common area is not overwritten when a new module is loaded.
Labeled commons are allocated immediately after the first object module in which they appear.

The output of the Disc Linkage Editor consists of:
- a load module
- error messages
- a listing and map (optionally).

Load Module
The executable program, output by the Linkage Editor and containing no more external references, is called *load module* and is written on disc on the temporary load file /L. If the user wishes to have this load module punched he may do so by specifying the PLD command (see CCI commands).
The load module is a random file, each sector containing 188 code words and a 12 word Relocation Bit Table. Each word of this table indicates per bit if a word is absolute or relocatable.
Bit 0 of the first word in the RBT points to the 1st code word (0 = absolute, 1 = relocatable). Bit 1 of the first word points to the 2nd code word. Bit 0 of the second word of the table points to the 17th code word etc.

The first four words of a load module contain information for the system loader and the Debug processor. The first link edited module starts from relative address 0008.



Error messages

Error messages are output when an error is encountered during processing. The table below shows whether the error is fatal (abort) or not.

Message	Output Unit	Abort?	Meaning
I O ERROR <file> <ssss>	T	Yes	Irrecoverable I/O error on file <file> with status <ssss> of the I/O unit.
BLK.COM.	P	No	Wrong optional blank common address.
BLK.DAT <name>	P	No	<name> is an unknown common block name used in a Block Data Subprogram.
DBL.DEF. <name>	P	No	<name> is defined more than once as an entry point or as the name of a common block.
INV.LGH. <name>	P	No	<name> is a common block name whose length exceeds the maximum length allowed.
UNS.EXT.	P	No	There are one or more unsatisfied external references. The load module may be executable when no references are made to its externals. The externals are listed in the map.
ABS.STR.	P	No	Absolute start address (ignored).
ERR.MOD.	P	No	A link-edited module has received an error flag from assembly or compilation.
NO STRT.	P	No	Wrong or not start address.
INV.IDT	T&P	Yes	Invalid IDENT record.
PRG.OVL.	T&P	Yes	Generated load module exceeds 32k words.
TBL.OVL.	T&P	Yes	Not enough space to link-edit these modules.
IDT.MIS.	T&P	Yes	IDENT record missing.
END MIS.	T&P	Yes	END cluster missing.
ERR.LKE	T&P	No	A non-fatal run has occurred during this link-edit run.
ABS.ADR.	T&P	Yes	An absolute address was read. The Disc Linkage Editor does not accept absolute addresses.

T = operator's typewriter
P = line printer

Listing and Map

When the parameter M in the LKE command is not specified, nothing will be printed on the operator's typewriter or line printer except for error messages.
When M has been specified the DLE prints the name of each link-edited module (including library modules), together with its relative beginning address, after completion of the link-edit run. The module names are listed, each on a new line, in the order of their occurrence.
Next a complete map of all entry points and common block names is given in an alphabetical order followed by their value. When a name is not defined yet four asterisks are printed instead of the value.
A letter indicating the type is printed behind the value.

A = Absolute
C = Common block
R = Relocatable
S = Internal symbol table
U = Undefined

Examples

This example shows which CCI commands may be used to assemble modules, to link and to execute them.

S:SCR	Clear the temporary files.
S:RDS	Read the modules to be assembled (from paper tape or cards) and place them in the /S file.
S:ASM /S	The modules are read from the /S file, assembled and placed in the /O file.
S:LKE N, M	The object output from the Assembler is read by the LKE. No scanning is requested and a map is printed. The load module is stored in the /L file.
S:RUN	The load module is executed.

The following example shows how to assemble a program stored in a library, execute it and store the load module in a library.

S:SCR	Clear the temporary files.
S:ASM <ident>	The name of this program is read from the library.
S:LKE N, M	The output from the Assembler is read by the LKE. No scanning is requested and a map is printed. The load module is stored in the /L file.
S:RUN	The load is executed.
S:KPF /L, <name>	The load module in the /L file is given the name <name>, and is stored.

PART 4 OVERLAY LINKAGE EDITOR

When a program has been assembled or compiled it consists of a (large) number of object modules which, together, may not yet form an executable program when there still are external references, i.e. references from one module to an other, to be matched.

Apart from linking of references requirements the program may have such a size that it will occupy most or more of the memory space available during execution. The Overlay Linkage Editor copes with both problems by linking all external references, if they can be matched, and by generating a segmented program according to the overlay technique. This overlay technique makes it possible to load into memory, at execution time, only those parts of the program which are required at a certain moment and which will be overlaid if their presence is no longer necessary.

Compared to a non-segmented program, the producing of a segmented program requires some extra words which are added to the load module. At execution time, however, a considerable memory space is gained. Using the Overlay Linkage Editor for segmented programs may therefore be useful only when the user program occupies most or more memory than being available.

The Overlay Linkage Editor operates in a P856M or P857M disc operating system under control of the DOM, DRTM or MAM monitor and occupies 6.5k words of memory.
The processor is also able to produce a non-segmented program. In that case the object modules offered to it on the input file for processing must be arranged in a different way.

1 Overlay Technique

This chapter contains a general description of the overlay technique and gives some definitions of terms.

The overlay technique is a programming technique which allows to reduce the memory space needed for program execution.
The program's object modules must be linked and organised in such a way that the program's modules are only loaded when their presence is required.

To obtain this goal an *overlay tree* structure can be designed which is the graphical representation of the program's organisation to meet the overlay requirements.
An overlay structure has as basis the *root* which is that part of the program which will always be in memory as it exercises the control of the program. The branches of the tree, called *paths,* constitute together with the root, the way along which the program is executed. Each path of the tree consists of one or more *segments* which may be built of one or more modules. The order in which the segments are placed along the path is determined by the user, depending on the program and the references segments make among each other.
The beginning of each segment is called *node.* The *level* of a segment in a path is the number of nodes between that segment and the root. In a path segments with a lower level are called *ascendants* and segments with a higher level are called *descendants.* A segment located in another path is called *exclusive.*

Example
A program consists of 10 modules which we will number a through j. Of these modules a and b form the root (segment 1).
In the program we can distinguish 6 paths and 9 segments (root included, of which segments 1, 4 and 9 are built of more than one module.

The 6 paths are:

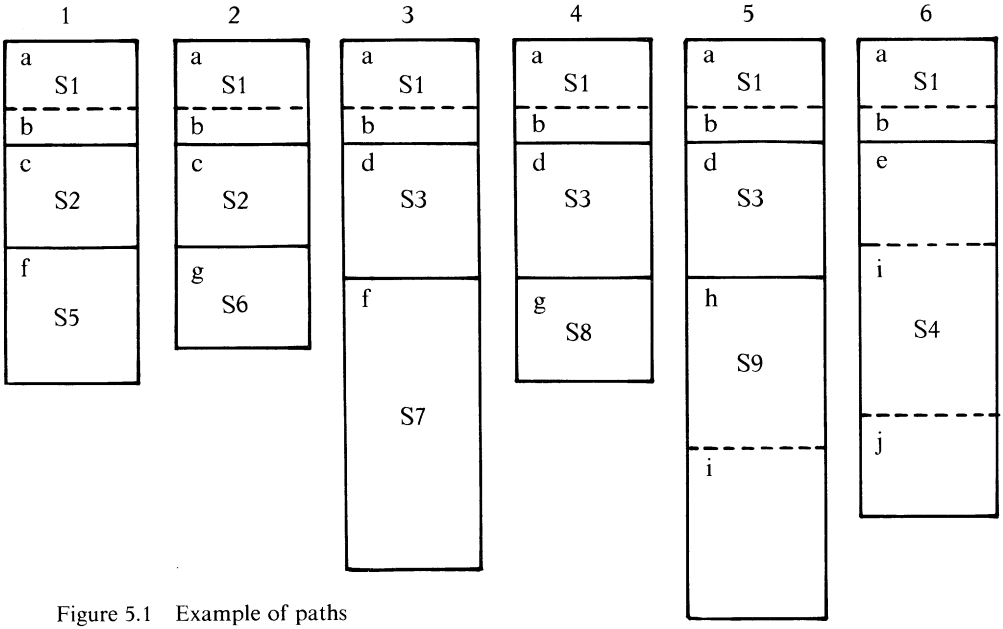


Figure 5.1 Example of paths

Of these 6 paths the following overlay tree may be built:

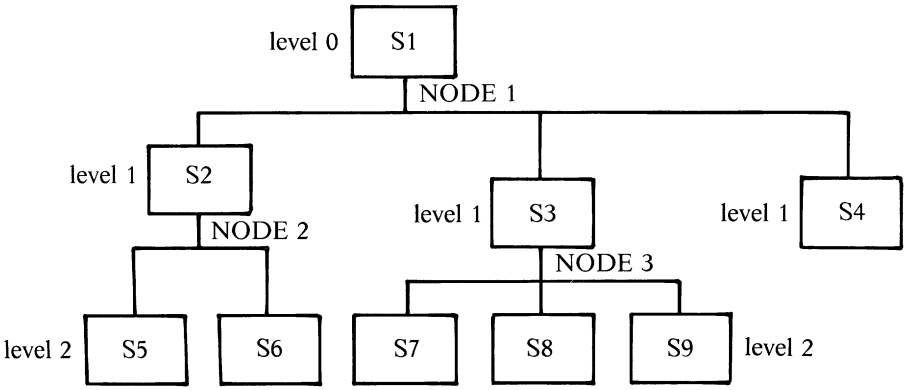


Figure 5.2 Example of Overlay Tree

In this example will, at execution time, first path 1 be executed and then path 2. Of the latter path are already in memory segments 1 and 2. Segment 5 of path 1 is no longer necessary and will be overlaid by segment 6. Segments 7 and 3 will next overlay segments 6 and 2 etc.

Though no specific programming requirements are necessary to have a successful link-edit and production of a segmented program certain rules should not be overlooked.

- segmented programs are not re-entrant as segments will be overlaid during a run. Only the root and blank common are never overlaid.
- it is not possible to return, by means of an RTN instruction or RETURN statement, from an exclusive segment if that segment has been called by a CF instruction or CALL statement. The reason is that the stack may not contain the right information for a proper return as the segment is overlaid by the exclusive.
- a Block Data Subprogram must be in the segment with the highest level using the common.

The program to be linked must be placed on the /O file. The modules may be called from the library by means of an INC command or modules may be assembled or compiled (ASM, HSF of FOR).
Since these modules may make references to other modules in the user or system library or both, the libraries have to be scanned by the processor to look for the missing references, i.e. when the user asks to do so in the OLE command, see below.
To facilitate the scanning of the user libary a library directory is created and kept up to date each time a module is kept in the library by a KPF /O command or deleted by a DEL /O command. The directory is placed in the user library by the system under the name OBDIR and consists of a random file of type UF.

Segmented program
To produce a segmented load module the program modules must be placed on the /O file according to the overlay design made.
The first segment on the file must be the root. Next the following segments must be loaded as follows:
— when several (exclusive) segments have the same immediate ascendant their common beginning location is called node. To define the node on the /O file the command NOD name must be given, where name in the command may consist of up to 6 alphanumeric characters specifying the name to be given to the node. This name is recorded as an ASCII record on the /O file and is used by the Overlay Linkage Editor, but the name is not included in the load module.
When several modules form a segment, as many INC, HSF, ASM or FOR commands may be given as there are modules, until a following NOD command.
Up to 128 segments may be specified for one run.
At the end of Chapter 4 an example is given how to proceed.
When all segments are on the input file the Overlay Linkage Editor must be called with the command OLE, whose syntax is:

OLE␣[N|S|U][,M][,[DE|DS]][,/ < adress >][, < entry name >]

where:

N = no library need to be scanned
S = the system library must be scanned
U = the user library must be scanned
default: the user and system library must be scanned (in this order)
M = a map of the overlay structure followed by a symbol table is printed on the listing device. In case a non-segmented program is to be produced the map has a different structure. See the example at the end of Chapter 4.
default: no map is printed

DE = the symbol table with all entries of the root is placed at the end of the root for debugging purposes

DS = the symbol table with only the internal symbol entry point list is placed at the end of the root for debugging purposes

default: no symbol table is added to the load module. However, a symbol table will be printed on the map

/<address> the user may specify the absolute hexadecimal address of a blank common.
default: the blank common is relocatable and is placed at the end of the program after the "region". See page 4-11.
In this way the blank common may be used as a system's common for communication between different tasks in a DRTM system.

<entry name> name of the start address defined as entry point in one of the modules of the *root.*
default: the program's start address will be the last start address defined in any module of the root, or the last start address encountered on the /O file (for non-segmented programs).
A start address in a module included during scanning is not taken into account.

Processing

The processor starts reading the whole input file /O into memory. All external references encountered during reading are placed in a symbol table, at the same time indicating whether the reference is absolute or relative.
The external references may consist of entry points or labelled commons.
The OLE now tries to match the references according to the overlay structure of the program and it takes the following rules into account:

- references are first looked for in the segment. If they cannot be found in the segment, the ascendants are searched and next the descendants.
  If a double definition was given in an ascendant the first one encountered is taken and a non-fatal error message is printed.
  If a reference is made to one or more descending segments the reference is defined the first time it is encountered. The external reference in that case is not replaced by the entry point's address but by the address of a link block which points to the segment loader.
- depending on whether the user has specified the relevant options in the OLE command, or uses the default value, the processor starts looking for the missing reference in the user library (U), the system library (S) or both (default). Each module, or file, of such a library has a directory containing all relevant information as given in clusters 2, 5, 6 and 7 concerning the entry points, externals or commons in the module or file.
  If the entry point is found in a library the module which contains the entry point is included in the program. If the module in which the entry point appears is referred to by more than one segment the module is included in the segment with the lowest level.

– If the external reference is not yet found after scanning one or both libraries the Overlay Linkage Editor starts looking for it amongst the exclusives.
As the referencing to exclusives might create stack problems a warning message is output.
When the reference is found in an exclusive segment the external is not replaced by the address of the entry point but by the address of a link block.
Should the entry point be present in more than one exclusive segment the first time the entry point is encountered is taken as the definition.

Processing of commons
Commons may be labelled or blank. The Overlay Linkage Editor processes the occurrence of blank or labelled commons in a different way.

Labelled commons
Labelled commons have a fixed length. They are allocated by the processor at the end of the segment in which they are referenced.
Consequently, they may be overlaid during a program run, but the initial values given by a Block Data Subprogram are reloaded each time the segment is loaded.
When a reference is made to a labelled common whose label is used in several segments, the common is allocated to the segment with the lowest level.

Blank common
The largest blank common encountered in the program is placed at the end of the program and is never overlaid.
The user must, however, take care not to destroy this area when he is using a Get Buffer request. The beginning address of this buffer must be pointing to a location after the last address of the blank common. See also the example on page 4.15.

When the user has given an absolute addres to a blank common in the OLE command the blank common is loaded at the address specified.

Load module
At the end of processing a segmented program is built.

When the load module is generated the object code from the modules is taken and relocatable words and external references are replaced by their real addresses. Moreover, some information is added in front of the root. See the description of the load module in Chapter 4.

The Overlay Linkage Editor adds each time a segment is loaded, a segment load block to the root which contains information of where the segment is to be loaded, its length and the sector of the disc where the segment can be found.
The last block is followed by segment loader which tests whether a segment has been loaded or not when the program is executed.

The output of the processor consists of:
- load module
- map (on option)
- symbol table (on option)
- error messages

LOAD MODULE

The load module is an executable program in object format. The module is output on the /L file. Each sector of the file contains 188 code words and a 12-word relocation table (RTB) of which:

- bit 0 of word 0 — 1 if the first word is relocatable; 0 if the first word is absolute
- bit 1 of word 0 — is associated with the second code word

  ⋮

- bit 0 of word 1 — is associated with the 17th code word

  ⋮

  etc.

The first code words of the load module, which are stored in the first locations of the program, have the following meaning:

- location 0 — **for non-segmented program**
  start address
  for segmented program
  start address is increased by 1 and points to a location in the root
- location 2 — **for non-segmented program**
  number of sectors occupied by the load module on disc
  for segmented program
  number of sectors occupied by the root
- location 4 — effective program length (blank common, if not given an absolute address in OLE command, included)
- location 6 — if DE or DS was specified in the OLE command this location contains the symbol table address
- location 8 — **for non-segmented program**
  first code word of the first module of the program
  for segmented program
  length of program area (which may be longer than effective program length. See description of REGION)

— location A and following

for segmented programs only

In location A is given the number of segments of the program, the root excepted.

The following n × 4 words contain n segment load blocks.

The last 4-word item is followed by the segment loader

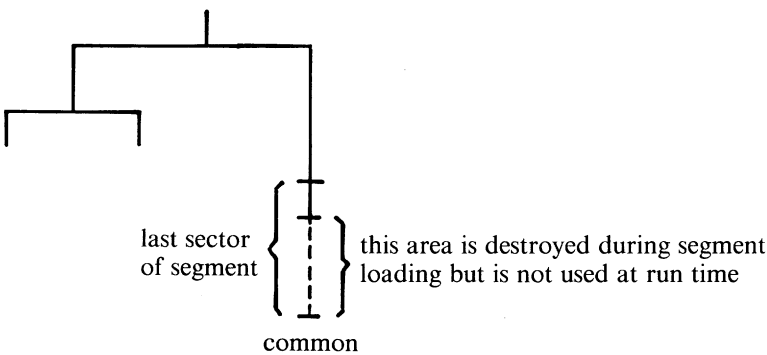
MAP AND SYMBOL TABLE

An example of a map is given at the end of this chapter.

START — start address of the program

LENGTH — length, in characters, of the longest path

REGION — since the segments are loaded per sector, an entire sector may therefore be loaded without the sector being completely filled with a segment. REGION is the length of the longest path as loaded from n sectors.

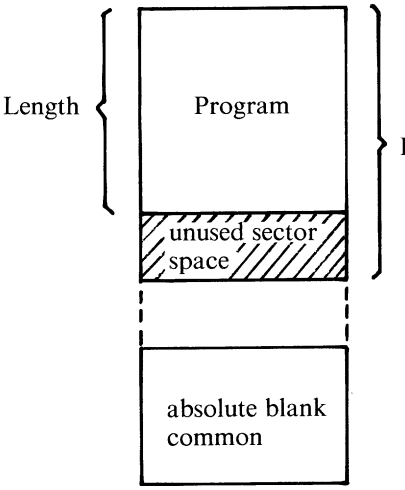


Example

The following example explains the difference between the length and region.

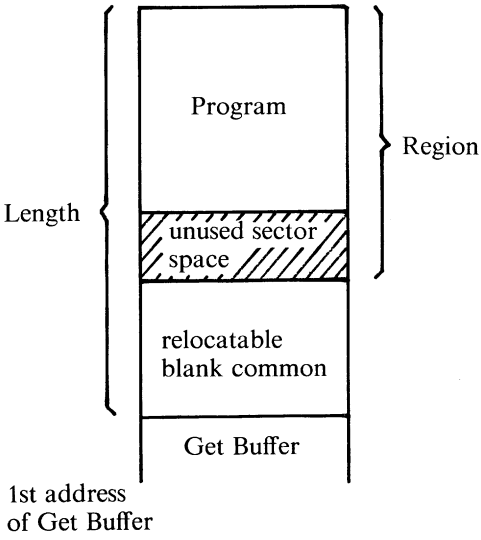
One time a program contains an absolute blank common and in the other example the program contains a relocatable blank common.

Absolute Blank Common



Here Length < Region, but it may happen that Length = Region if all sectors were entirely used

Relocatable Blank Common



Next the segments are printed in ascending level order.
where:

segment #	number of the segment in the overlay tree
address	address of the segment in memory
sector #	number of the sector in which the segment is written on /L
ascendant #	number of the segment's immediate ascendant. For the root this is always # FF.

Then the ident(s) of the module(s) in the segment are printed plus their module's address.
The list of segments is followed by the symbol table. All entry points and common blocks, belonging to a segment, are listed in an alphabetic order. The symbol table is built of the following items over 4 rows:

type segment address name

where:

type = A absolute entry point
B absolute address of blank common as given in OLE command
C relocatable blank or labelled common
D symbol table entry point
E relocatable entry point

segment = number of the segment in which the symbol appears
address = address of the symbol
name = entry point name. If the same name is defined in several segments the name is printed each time it is encountered but with a different segment number.

ERROR MESSAGES

During and after processing error messages may be printed which may (fatal error) or may not (non-fatal error) influence the processing and the output of a correct load module.
At the end of processing the number of errors are printed if there were any, followed by the error message(s).

Fatal errors

Fatal errors cause the /L file to be scratched and no load module is produced. Fatal errors are printed on the line printer and on the operator's typewriter.

Message
CORE OVERFLOW (not enough space for the processor)
DIRECTORY AND SYSTEM LIBRARY NOT CONSISTENT
DIRECTORY AND USER LIBRARY NOT CONSISTENT
END MISSING (EOF FOLLOWING IDENT)

END MISSING (NOD FOLLOWING IDENT)
END MISSING (2 CONSECUTIVE IDENT)
FIRST ROOT RECORD IS EOF
IDENT MISSING (END FOLLOWING NOD)
IDENT MISSING (EOF FOLLOWING NOD)
IDENT MISSING (FIRST OF ROOT)
IDENT MISSING (2 CONSECUTIVE NOD)
IDENT OR NOD RECORD MISSING OR INVALID
I/O ERROR <ECB0> <ECB8>
NOD NOT ALLOWED BEFORE THE ROOT
PROGRAM LENGTH EXCEEDS 32k (the longest program unit > 32k)
2 CONSECUTIVE NOD

Non-fatal errors

Non-fatal errors are in fact warning messages for the user. The error is printed on the line printer.
The processor continues processing but the produced load module may or may not be executable.
The hexadecimal number of errors is printed on the line printer and operator's typewriter after processing.

Messages

ABSOLUTE ADDRESS IN MODULE <name> SEGMENT <number>
ABSOLUTE START ADDRESS IN MODULE <name>
DOUBLE DEFINITION ON <name>
(the first definition is taken)
ERROR IN MODULE <name>
EXCLUSIVE REFERENCE FROM SEGMENT <number> TO <name> IN SEGMENT <number>
NO START ADDRESS
REF. TO UNSATISFIED EXTERNAL <name> IN SEGMENT <number> AT ADDRESS <number>
(the address is relative to the beginning of the segment)
UNDEFINED START ADDRESS NAME
(the name specified in the OLE command is not defined in the root)
<number> UNSATISFIED EXTERNAL REFERENCES
(<number> is in hexadecimal. The symbol table indicates which external references could not be matched by printing an asterix, as follows:

```
                        ***   SYMBOL TABLE   ***

E 00 00EE $ROOTF   E 01 017A %SEGF1   E 02 01BC $SEGF2   E 03 01BC $SEGF3
E 04 017A %SEGF4   E 05 028C $SEGF5   E 06 01BC $SEGF6   E 00 00EE %ROOTF
E 01 0174 %SEGF1   E 02 01B6 %SEGF2   E 03 01B6 %SEGF3   E 04 0174 %SEGF4
E 05 0278 %SEGF5   E 06 01B6 %SEGF6   * ** **** F:CL     * ** **** F:ERR3
* ** **** F:ERR4   * ** **** F:ER10   * ** **** F:ER11   * ** **** F:ERLK
A 00 00C0 F:FCT    * ** **** F:L2     * ** **** F:S2     * ** **** F:ST
* ** **** F:SY     E 01 015A SEGF1    E 02 019C SEGF2    E 03 019C SEGF3
E 04 015A SEGF4    E 05 0216 SEGF5    E 06 019C SEGF6
```

Example:
The following FORTRAN program consists of 7 modules of which module ROOTF is the root.

```
      IDENT ROOTF
      COMMON N
      N=0
      CALL SEGF1
      CALL SEGF4
      END
```

```
      IDENT    SEGF1
      SUBROUTINE SEGF1
      COMMON M
      M=M+1
      WRITE(2,100)M
      CALL SEGF2
      CALL SEGF3
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
      END
```

```
      IDENT    SEGF2
      SUBROUTINE SEGF2
      COMMON K
      K=K+1
      WRITE(2,100)K
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
      END
```

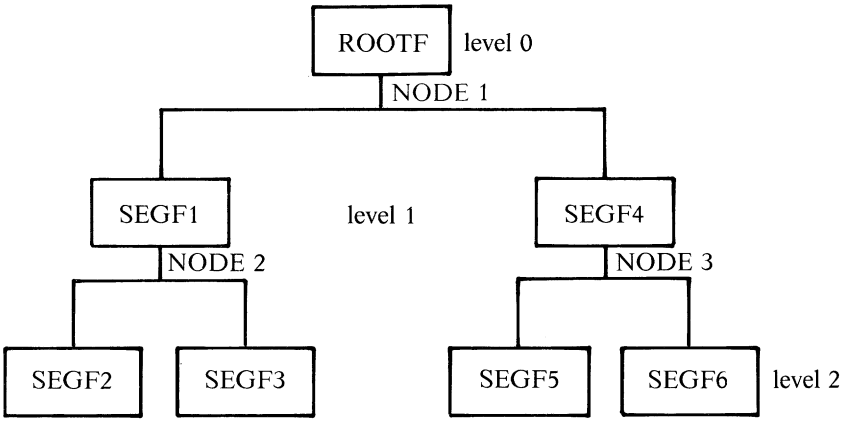
```
      IDENT    SEGF3
      SUBROUTINE SEGF3
      COMMON L
      L=L+1
      WRITE(2,100)L
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
      END
```

```
      IDENT    SEGF4
      SUBROUTINE SEGF4
      COMMON M
      M=M+1
      WRITE(2,100)M
      CALL SEGF5
      CALL SEGF6
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
      END
```

```
      IDENT    SEGF5
      SUBROUTINE SEGF5
      COMMON K
      DIMENSION I(40)
      K=K+1
      DO 10 J=1,40
10    I(J)=J
      WRITE(2,100)K
      WRITE(2,200)I
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
200   FORMAT(1X,'TABLEAU I :'//5(1X,8(I2,1X)))
      END
```

```
      IDENT    SEGF6
      SUBROUTINE SEGF6
      COMMON L
      L=L+1
      WRITE(2,100)L
      RETURN
100   FORMAT(1X,'SEGMENT ',I1)
      END
```

Of this program the following overlay tree may be built:



Supposing that all modules are in the user object library, where they have been kept after HSF or FOR, the following procedure can be used to load all the modules on the /O file.

```
INC ROOTF
NOD N1
INC SEGF1
NOD N2
INC SEGF2
NOD N2
INC SEGF3
NOD N1
INC SEGF4
NOD N3
INC SEGF5
NOD N3
INC SEGF6
```

The sequence may be followed by:

OLE S,M,/3000

Then the map is printed:

```
                START = 00C6    LENGTH =167E    REGION =17A4
                     ***   OVERLAY STRUCTURE   ***

                          ***   LEVEL #    0   ***

    SEGMENT # 00        ADDRESS = 00C4   SECTOR # 0000  ASCENDANT # FF
ROOTF   00C4       F:MS     00F2       F:IOWS  0204

                          ***   LEVEL #    1   ***

    SEGMENT # 01        ADDRESS = 0228   SECTOR # 0002  ASCENDANT # 00
SEGF1   0228       F:SEQW   0292       F:SEQU  02A4       F:PREP  03DA
F:EXCL  0434       F:VARP   0714       F:AWOS  07A4       F:LWOS  070E
F:OUTN  081C       F:FLOC   0994       F:ICIC  0B84       F:COPY  0BAC
F:BAZP  0BCC       F:IORC   0C06       F:IOCS  0C84       F:STAT  0CB2
F:ASCI  0D8C       F:DI     0DA4       F:DD    0EA8       F:DM    105C
F:DN    1108       F:T3     1236       F:FX    1266       F:FL    12F4
F:RP    1306       F:TLN    130A       F:TRA   143E

    SEGMENT # 04        ADDRESS = 0228   SECTOR # 0011  ASCENDANT # 00
SEGF4   0228       F:SEQW   0292       F:SEQU  02A4       F:PREP  03DA
F:EXCL  0434       F:VARP   0714       F:AWOS  07A4       F:LWOS  070E
F:OUTN  081C       F:FLOC   0994       F:ICIC  0B84       F:COPY  0BAC
F:BAZP  0BCC       F:IORC   0C06       F:IOCS  0C84       F:STAT  0CB2
F:ASCI  0D8C       F:DI     0DA4       F:DD    0EA8       F:DM    105C
F:DN    1108       F:T3     1236       F:FX    1266       F:FL    12F4
F:RP    1306       F:TLN    130A       F:TRA   143E

                          ***   LEVEL #    2   ***

    SEGMENT # 02        ADDRESS = 1496   SECTOR # 000F  ASCENDANT # 01
SEGF2   1496

    SEGMENT # 03        ADDRESS = 1496   SECTOR # 0010  ASCENDANT # 01
SEGF3   1496

    SEGMENT # 05        ADDRESS = 1496   SECTOR # 001E  ASCENDANT # 04
SEGF5   1496       F:IA     15C6

    SEGMENT # 06        ADDRESS = 1496   SECTOR # 0020  ASCENDANT # 04
SEGF6   1496

                          ***   SYMBOL TABLE   ***

A 00 3000          E 00 00E8 $ROOTF   E 01 0286 $SEGF1   E 02 14EC $SEGF2
E 03 14EC $SEGF3   E 04 0286 $SEGF4   E 05 15B8 $SEGF5   E 06 14EC $SEGF6
E 00 00E8 %ROOTF   E 01 027C %SEGF1   E 02 14E2 %SEGF2   E 03 14E2 %SEGF3
E 04 027C %SEGF4   E 05 15A2 %SEGF5   E 06 14E2 %SEGF6   E 01 0D8C F:ASCI
E 04 0D8C F:ASCI   E 01 07A4 F:AW     E 04 07A4 F:AW     E 01 0510 F:CL
E 04 0510 F:CL     E 01 076C F:CLIO   E 04 076C F:CLIO   E 00 020A F:CLOR
E 01 0BAC F:COP    E 04 0BAC F:COP    E 05 165C F:DI     E 01 0EC4 F:DD
E 04 0EC4 F:DD     E 01 0DD6 F:DI     E 04 0DD6 F:DI     E 01 1076 F:DM
E 04 1076 F:DM     E 01 1108 F:DN     E 04 1108 F:DN     E 00 01CA F:ER
E 00 01B8 F:ER03   E 00 01BC F:ER04   E 00 01C0 F:ER10   E 00 01C4 F:ER11
E 00 01C8 F:ERLK   A 00 0000 F:FCT    E 01 12F4 F:FL     E 04 12F4 F:FL
E 01 0994 F:FLOC   E 04 0994 F:FLOC   E 01 126C F:FX     E 04 126C F:FX
E 01 0B84 F:ICI    E 04 0B84 F:ICI    E 05 15D8 F:ID     E 05 15C6 F:IM
E 01 0BEA F:IOAS   E 04 0BEA F:IOAS   E 01 0BCC F:IOB    E 04 0BCC F:IOB
E 01 0C84 F:IOCS   E 04 0C84 F:IOCS   E 00 0208 F:IOCV   E 01 0BE2 F:IOP
E 04 0BE2 F:IOP    E 01 0C06 F:IORC   E 04 0C06 F:IORC   E 01 0C2A F:IORQ
E 04 0C2A F:IORQ   E 01 0BDE F:IOZ    E 04 0BDE F:IOZ    E 05 15EE F:IX
E 01 0518 F:L1     E 04 0518 F:L1     E 01 051C F:L2     E 04 051C F:L2
A 00 0002 F:LPFC   E 01 07DE F:LW     E 04 07DE F:LW     E 01 0D78 F:MESS
E 04 0D78 F:MESS   E 01 081C F:OUTN   E 04 081C F:OUTN   E 00 0162 F:PA
E 01 03DA F:PREP   E 04 03DA F:PREP   E 01 0782 F:RACL   E 04 0782 F:RACL
E 00 02A6 F:RECV   E 01 0D6C F:RLWS   E 04 0D6C F:RLWS   E 01 1386 F:RN
E 04 1386 F:RN     E 01 1308 F:RP     E 04 1308 F:RP     E 01 130C F:RS
E 04 130C F:RS     E 00 0204 F:RSEV   E 05 162C F:S2     E 05 1630 F:S4
E 05 1634 F:S6     E 01 02A4 F:SEQU   E 04 02A4 F:SEQU   E 00 0136 F:ST
E 01 0CB2 F:STAS   E 04 0CB2 F:STAS   E 01 0CCA F:STAT   E 04 0CCA F:STAT
E 01 0798 F:SUB    E 04 0798 F:SUB    E 01 0292 F:SW     E 04 0292 F:SW
E 01 1270 F:T1     E 04 1270 F:T1     E 01 1244 F:T3     E 04 1244 F:T3
E 01 130A F:TLN    E 04 130A F:TLN    E 01 143E F:TRA    E 04 143E F:TRA
A 00 0001 F:TYFC   E 01 0714 F:VAR    E 04 0714 F:VAR    E 01 0242 SEGF1
E 02 14B0 SEGF2    E 03 14B0 SEGF3    E 04 0242 SEGF4    E 05 1528 SEGF5
E 06 14B0 SEGF6
```

In this example we can see that segment 1 is overlaid by segment 4 and the segment 2 by segments 3, 5 and 6.

If not all modules were kept in the user library, also the following sequence is accepted, e.g.:

```
HSF ROOTF
NOD N1
INC SEGF1
NOD N2
HSF SEGF2
NOD N2
INC SEGF3
NOD N1
INC SEGF4
NOD N3
HSF SEGF5
NOD N3
INC SEGF6
OLE M
```

The contents of the /L file may be kept with a KPF command.
If a KPF /O is given for the /O file containing NOD's, the modules on this file are catalogued without taking care of the NOD's.

Non-segmented programs
If the user wishes to produce a non-segmented program, which usually are small programs, the Overlay Linkage Editor operates as if all modules, placed on the /O file by means of ASM, INC, FOR or HSF commands but no NOD commands are used to separate the modules, are a segment of level 0. No link blocks or segment loader are added to the load module.

PART 5 LINE EDITOR

User and system programs must be maintained regularly. This means that new data has to be included and old data to be deleted.
The Line Editor allows to update a source program or user data on a line level as well as on a character level.
The edited output file is written as a temporary /S file. If /S already exists a new assignment is made and the old /S is overwritten.
In order to keep the updated file the user must not forget to terminate updating with a KPF command.

Editing sequential source files or text by means of the Line Editor takes place in two phases:
– definition phase
– execution phase.

Definition phase
When the Line Editor is called with the LED control command it is flagged as being in definition phase. At this phase, which lasts until an Execution message is given, the user may specify by means of the !!CH message, which character strings, being the same and appearing throughout the module, have to be changed in another character string no matter how many times this particular character string to be updated is encountered. This message which may be typed as many times as necessary with different parameters, is pre-stored in memory. The relevant Definition phase messages are executed before the relating modified file is output.
When the user requires the immediate listing of all lines with a specific character string he may type in the !!LS message.

The definition phase is closed i.e. no more Definition messages may be entered as soon as the first Execution message is typed in.

Execution phase
Messages typed in at this phase which is started by any of the Execution messages, are immediately executed the moment they are typed in and LF CR is given. These messages allow to insert any number of lines, or replace a character string in a line.

LINE EDITOR COMMAND

Before the user may start editing his files he must call the Line Editor by means of the following CCI command:

LED␣<name>⌊⌈,<file code1>⌈,<file code2>⌉⌉|⌈,/S⌈,<file code2>⌉⌉⌋⌈,XX⌉
where:

<name>	name of source module or user data file to be edited
<file code1>	output file code for edited file. If this parameter is used the type of file is implicitly UF.
<file code2>	file code from which Editor messages are read.
<XX>	2 alphanumeric characters which will be used instead of!! preceding each command.

Note: when working with UF type of files all records are output with 80 characters

If no parameter follows <name> it is assumed that a source program is to be updated and that commands are read from /E0. When <file code 2> is assigned to the operator's typewriter the characters L: are printed before reading any record.

When this command is accepted the Line Editor will type out L: on the typewriter log. The user may reply by any of the Control Messages he considers necessary.
He must edit his module in an ascending order of line numbers.

CONTROL MESSAGES

Line number: decimal number ranging from 0 to 9999 included.
A character string consists of a number of characters, commas and blanks included. By delimiting it by two $ $ signs preceding and terminating the characters the user may determine the length of the character string in his control message.

Example:
„The user must not forget" . . .
can be changed in „should keep in mind" . . .
as follows:
!!RE␣<line number>,$ $must not forget$ $should keep in mind$ $

Definition phase message

!!CH␣$ $character string 1> $ $ <character string 2> $ $

This message is used to have <character string 1> replaced by <character string 2> wherever string 1 appears in the module no matter how many times it is encountered.
The message is pre-stored in memory but not yet executed. More than one !!CH message may be typed in at definition phase.

NOTE: Be aware that, when using !!RE later on, the statement to be changed by !!RE may already have been changed by !!CH.

!!LS␣$ $ <character string> $ $

This message causes an immediate listing of all lines (of the input file) containing this character string on file code /02. The operation is terminated when the :EOF mark is encountered.

Execution phase messages

!!JN␣[<line number>], <name>, <line no a> [, <line no b>]

The lines indicated by <line no a> thru <line no b>, both included, of the file with name <name> on the Auxiliary file are inserted immediately after the <line number> of the current input.
If <line number> is omitted the lines are inserted after the current line of the main input file.
Records of the Auxiliary file may be altered by the !!CH commands, if any.

!!RE␣ <line number>, $ $ <character string 1> $ $ <character string 2> $ $

This message is used to replace in the line with line number <line number> <character string 1> by <character string 2>.
If <string 2> is longer than <string 1> the last characters of the input line are truncated.
If <string 2> is shorter than <string 1> blanks are added to make a record of 80 characters.
It is possible to write new lines immediately after this line. An !!IL command for this line number is not possible.

!!DL␣ <line number 1> [, <line number 2>]

The line, specified by <line number> is deleted or all lines from <line number 1> to <line number 2> (both included) are deleted.
All the following lines, if not beginning with !! are inserted after the deleted line. Deleted lines are listed on /02 file and are not altered by !!CH commands.

!!IL␣[<line number 1>]

This message allows to insert records after the current line, (parameter <line number> absent), or to insert lines after the specified line number.
The Package continues inserting lines until the next control message.
Inserted Lines are listed on /02 file and are not altered by !!CH commands.
An !!IL command cannot be given for the same line number as used by the preceding !!RE command.

!!EN

This control message terminates the updating. The remaining records of the input file are copied on the output file. They may be changed by !!CH command. This message or the message !!AB must be the last control message.

!!AB

This control message aborts the current update and the output file is scratched. The user may gain control on the operator's typewriter to enter a correction (only if not under batch processing mode).

The last command to be typed in must be the KPF command in order to keep the updated file.

Message	Meaning
FILE NAME ERROR	The specified name in the message contained an error.
FILE NAME MISSING	The specified name cannot be found on this disc.
INPUT FILE CANNOT BE ASSIGNED	This error message is followed by a message explaining the error.
/S CANNOT BE ASSIGNED	This error message is followed by a message explaining the error.
INVALID FILE CODE	The file code specified does not belong to the input device from which the control messages are input.
FILE CODE NOT ASSIGN	The file code of the message input device must have been specified beforehand by means of ASG.
TOO MANY PARAMETERS	Too many parameters specified.
DSK INPUT ERR, UPD ABORTED	Line Editor cannot read from disc.
DSK OUTPUT ERR, UPD ABORTED	Line Editor cannot write onto disc.
UNKNOWN COMMAND, [TRY AGAIN]	The introduced control message is not accepted as it was not one of those described under Processing.
I/O ERR ON LAST RECORD, [TRY AGAIN]	An I/O error occurred.
SEQUENCE ERR, [TRY AGAIN]	The line numbers in the control messages are not in ascending order.
SYNTAX ERR, [TRY AGAIN]	The introduced control message or the newly typed in line, contained a syntax error.
AUX INPUT CANNOT BE ASSIGNED [TRY AGAIN]	The auxiliary file used in JN command cannot be assigned.
CMND NOT ALLOWED IN EXE MODE [TRY AGAIN]	This command cannot be used in the execution phase.

TABLE O'FLOW, [TRY AGAIN]	The character string table is overflowing.
EOF, UPD TERMINATED	The :EOF mark has been encountered on the input source file before reaching the specified line, thus terminating the update process.
EOF IN AUXI INPUT	The :EOF mark has been encountered from the Auxiliary Input. The JN command is terminated but the operator continues.

When the message TRY AGAIN is printed the user has the possibility of correcting the previous command or data record from /01. If CR is typed in, the input is resumed from the normal input command file.

In batch processing mode the message TRY AGAIN is not printed and the job is terminated.

PART 6 DEBUGGING PACKAGE

The Debugging Package is used to check and test programs written in Assembly Language.
After the user's source program has been assembled the resulting object program, with its symbol table if a STAB directive has been used, must be loaded on the /L file by the Linkage Editor.
The Debugging Package loads a program from a catalogued Load module file. The syntax of the Debug calling command is: DEB␣[<name>], where <name> is the name of the program to be debugged.

Once the Package is called the user may set breakpoints where the Package will suspend execution of the user program to be tested and await further commands. This allows the user to check his program section by section. He may examine the contents of any memory location and modify those lying within the program's boundaries or he may test the contents of register A1 thru A14 before starting the execution or during breakpoint halts.

The Debug Package is not reentrant, therefore it is not possible to test a main program and its scheduled labels simultaneously. It is allowed to define breakpoints in scheduled labels.

By using the Debugging Package the user is able to test a program rapidly and should a bug be detected alterations are quickly made after which the corrected section may be executed.

In Part I was described how the programmer could keep a module's internal symbol table by including the STAB directive.
The inclusion of that table allows the user, during debugging, to address locations by using the symbols pertaining to the locations.
In the LKE command the user was given the option DE or DS permitting him to include in his linked modules either solely the internal symbol tables (DS) or the symbol tables together with entry points (DE).
The latter feature permits to debug several related subroutines simultaneously.
If a symbol is declared as entry point in a module the symbol is referenced in a different way as the internal symbol (see Chapter 2 Parameter syntax).

Locations in memory may be addressed in three ways:
- absolute
- relative
- symbolic

A location is addressed **absolute** as follows:
1) take the absolute load address printed on the typewriter log after having called the Debugging Package.
2) add the relative address of the location as printed on the Linkage Edit map.
   or
3) step two can be replaced by the following:
   - add the relative address of the location as printed on the assembly listing of the program to be debugged.
   - add 8 i.e. the first four words of the load module containing information for the Debugging Package must be taken into account by the first module at link-edit time.

The **relative** address of a memory location is found as follows:
1) - take the location's relative address as appearing on the assembly listing
   - add 8 (see absolute addressing)
   or
2) - take the relative address as printed on the map after link-edit

Symbolic addressing takes place as follows:
DS option specified
- take the symbol table's name and specify a symbol which may be followed by a positive or negative decimal number indicating a displacement relative to that symbol.

DE option specified
- specify an entry point followed by a positive or negative decimal number
- or the function as specified under DS option.

Refer to section 'Parameter syntax' for more details.

On-line/Off-line

The Debugging Package allows on-line as well as off-line operation:

On-line: a Debug command is executed immediately after having terminated with LF CR. All commands described in chapter 2, except the IF command, are accepted in this mode which begins immediately after having called the Package or when a breakpoint terminated by a RT command has been executed.

Off-line: mode which is entered when a breakpoint is defined. The user may now type one command or a number of commands which will be executed when the user program is started and reached the specified breakpoint. The string of commands pertaining to a breakpoint must be terminated by either a GØ or a RT command. The user program's execution is suspended during processing of the commands.

Input/Output

Input

Input to the Debugging Package consists of either commands or of data. Commands are usually input from the operator's typewriter when the Package requests input after having typed D: on the typewriter log.

One command, CI, permits to read commands from either the card reader or the punched tape reader.

The command RE allows to read data from a specified device. This is particularly useful when the contents of buffers have to be changed or filled. The file code of the respective devices must have been assigned before calling the Package.

If commands are read from a device other than the typewriter and the input command is erroneous, this command with error indication and error message is printed on the typewriter log. The correct command may be typed in from the operator's typewriter after D: has been printed on the log.

The following commands are read again from the assigned input device.

Output

The output is normally directed to and printed on the typewriter log. However, the output may be directed to another output device by means of the CØ command provided the file code was correctly assigned before calling the Debugging Package.

Commands typed in from the typewriter are copied on the line printer.

Note: If the Monitor aborts the program to be debugged the Debugging Package keeps control. The Program Status Word is printed on the typewriter log and the line printer, followed by the relative abort address and the print-out of the contents of 14 registers. On the line printer, or on the typewriter log if the output was directed to TY, a memory dump takes place of the relating area.

A command consists of two ASCII characters which may be followed by a blank and one or more parameters. Parameters are separated by a comma.
If a command is terminated by a full stop another command (or commands) may follow on the same line but may not continue on the next line.
Each command, or the last command of a line if more than commands are specified on one line, must be terminated with LF CR.

Parameter syntax:

< memory reference > ::= < absolute address >
< relative address >
< symbolic address >

where:

< absolute address > ::= / < up to 4 hexa digits >
In IF command: M␣ < hexa number >

< relative address > @ < up to 4 hexa digits >

< symbolic address > 1. when DS option is specified:
$ < symbol table name > & < label > ± < decimal number >
2. when DE option is specified
$ < symbol table name > & < label > ± < decimal number >
$ < entry point > ± < decimal number >

< register > R < two digit decimal number >
< constant > / < up to 4 digit hexa number >

BREAKPOINT DEFINITION

syntax: AT␣ < memory reference >

The breakpoint facility provides a means of suspending the execution of a user program at any point. To set a breakpoint the user types AT followed by the absolute, relative or symbolic address of the word where he wants the program to stop.

Once the breakpoint is set it switches the Package to the off-line mode thus permitting to define a command or string of commands which are to be executed when the running user program reaches the location specified in the breakpoint.
The breakpoint's absolute address is printed when it is executed: BP: absolute address.

Breakpoints are kept in a table. The maximum number of breakpoints allowed in this table is 8. Once a breakpoint is executed or when the user does not consider a breakpoint necessary anymore it may be deleted from this table by the command DB.
This permits to use more than 8 breakpoints in a debug run though only 8 breakpoints can be present in the table at the same time.

Addresses specified in breakpoint defination need not to be in an ascending order, so the user may first define an address at the end of the program and then one at the beginning if he wishes to do so.

During a debug run a breakpoint may be defined only once, unless the breakpoint is deleted.
When a breakpoint is reached the instruction to which it points is executed.
The last command of a string of commands pertaining to a breakpoint must be either GØ or RT, concluding this breakpoint definition.

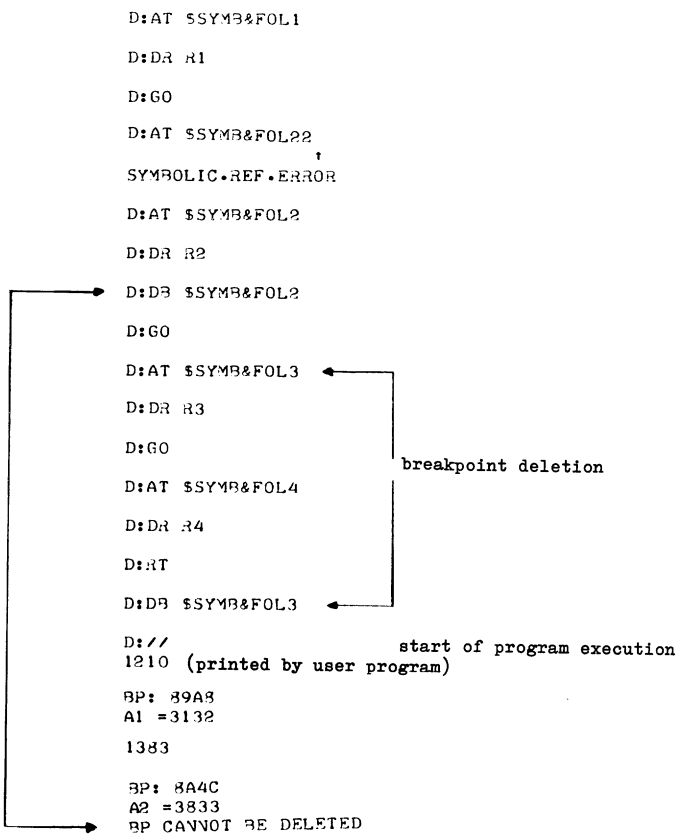
Restrictions

The breakpoints defined may not:
- be modified by the program
- refer to DATA defined text (the BP is not executed)
- refer to a LKM (or MLK) instruction
- refer to an address defined in an EX, EXK or EXR instruction

DELETE A BREAKPOINT

syntax: DB␣ <memory reference>

This command is used to delete a breakpoint and the commands defined to be executed at this breakpoint.
A breakpoint can be deleted from the breakpoint definition table at any time except when it is being executed.



Example 6.1

DUMP MEMORY

syntax: DM␣ <memory ref 1>,<memory ref 2>

Through this command the user may examine the content of a memory area from and including <memory ref 1> thru <memory ref 2>.
The dump takes place on the operator's typewriter unless otherwise specified by the CØ command beforehand.
The dump is presented as 8 words per line. Each line is preceded by an absolute address (multiple of /10). The last line of the dump is filled up to the eight word of that line, with words immediately following <memory ref 2>.

The underlined values are in the requested area.

```
D:DM /6511,/6528
6510  7D84  F4A5  8B22  F6DE  2020  5245  5345  5256      "   RESERV
6520  4544  2020  4E55  4D42  4552  2055  4E4B  4E4F  ED  NUMBER UNKNO
```

Example 6.2

WRITE MEMORY

syntax: WM␣ <memory ref>, <constant 1>[, <constant 2> ..., <constant n>]

This command permits the user to substitute the content of a memory location by an other value or, if more constants are specified, as many memory locations, from <memory ref> on, as there are constants specified.
The locations' contents must be within the program boundaries.

```
D:DM /6D08,/6D12.WM /6D08,/1234,/5678.DM /6D08,/6D0A
6D00  444F  4320  5245  5620  5041  4745  2033  3520  DOC REV PAGE 35
6D10  0D0A  3131  3632  3320  5038  3535  4D20  5359    11623 P855M SY
6D00  444F  4320  5245  5620  1234  5678  2033  3520  DOC REV  4V  35
```

Example 6.3

DUMP REGISTER

syntax: DR␣ <register>[, <register n>]

This command dumps, in hexadecimal format, the contents from 1 to n registers. <register> may be any of the user registers A1 thru A14 (see parameter syntax).
If no parameters are specified the contents of all registers, except for the P and A15 registers, are dumped.
If only one parameter is specified the contents of that register is dumped.
If both parameters are present the contents of those registers, both specified included, are dumped.
The dump is given on the typewriter log unless otherwise specified in the CØ command.

```
D:DR R1
A1 =0000

D:DR R1,R5
A1 =0000  A2 =0000  A3 =0000  A4 =0000  A5 =0000

D:DR R15
      ↑
PARAMETER ERROR
```

Example 6.4

WRITE REGISTER

syntax: WR␣ <register>, <constant 1>[, <constant 2> . . ., <constant n>]

The content of the specified register will be changed to the value of the first constant or, if more constants are specified, a number of consecutive registers is loaded with the values of the same number of consecutive constants. The first register of the range is the register specified in this command.

```
D:DR R1,R4
A1 =0000  A2 =0000  A3 =0000  A4 =0000

D:WR R1,/0011,/0022

D:DR R1,R3
A1 =0011  A2 =0022  A3 =0000

D:DR R1,R4
A1 =0011  A2 =0022  A3 =0000  A4 =0000

D:WR R3,/0033,/0044

D:DR R1,R4
A1 =0011  A2 =0022  A3 =0033  A4 =0044
```

Example 5.5

```
D:AT $SYMB&FOLI

D:DR R1,R4

D:WR R1,/0066,/0077,/0012

D:DR R1,R4

D:RT

D://
1210

BP: 88D6
A1 =3132  A2 =3130  A3 =3132  A4 =3130
A1 =0066  A2 =0077  A3 =0012  A4 =3130

D:
```

Example 6.5

CHANGE INPUT DEVICE

syntax: CI␣/ <file code>

When this command is given all further commands are read from the device with the specified file code.
Input devices other than the teletype may be the card reader or the punched tape reader. Their file codes must be assigned before the Package is called. In order to return the input of commands to the operator's typewriter the command CI␣/E0 must be given.

```
S:ASG /E1,PR20

S:DEB DOC4
DEBUG IS GOOD FOR YOU
YOUR LOAD ADDRESS = 6504

D:AT $SYMB&FOL2

D:CI /E1

D:RT

D://
1383


BP: 89FE
A1 =3133  A2 =3833  A3 =3133  A4 =3130  A5 =0000
A1 =3133  A2 =F1F1  A3 =1234  A4 =2345  A5 =6161

D:
```

```
DR R1,R5
WR R2,/F1F1,/1234,/2345,/6161
DR R1,R5
CI /E0

This was punched on tape
```

Example 6.6

CHANGE OUTPUT DEVICE

syntax: CØ␣/ <file code>

This command directs the output, from the time the command is read, to the output device with the specified file code.
This file code must have been assigned before the Package is called.
If output is to be returned to the operator's typewriter the command CØ with the typewriter file code must be given.
Whether the command is given on-line or off-line does not affect its operation i.e. the output device remains the specified output device until a new CØ command is given.

RETURN TO ON-LINE MODE

syntax: RT

This command concludes a breakpoint definition.
When the user program runs it will stop at the address defined by the breakpoint of which RT is the termination. The Debugging Package resumes control and types out D: on the typewriter log thus switching to on-line mode.
The user may now type in new commands. In this way it is possible to react immediately on the results of a breakpoint execution.

CONTINUE EXECUTION

syntax: GØ␣[< memory reference >]

This command concludes, as RT, a breakpoint definition. The difference of both commands is clear when the user program is executed and the breakpoint belonging to GØ is reached.
The breakpoint's absolute address is printed or punched and commands belonging to the breakpoint are executed. When the GØ command is read control is not returned to the operator's typewriter, as with RT. The execution of the user program continues until a new breakpoint is encountered.
In this case the user has not be possibility to react immediately on a breakpoint's execution results.
If < memory reference > is specified the user program will continue at the specified address which must be within the program's boundaries. If < memory reference > is not specified the execution resumes at the address following the breakpoint.
In example 6.7 two breakpoints are specified. One terminated by GØ and the other one by RT. The first breakpoint is printed on the typewriter log as the CØ command was not yet read. All other output is printed on the line printer log. After RT the user may input another command.

CONDITIONAL EXECUTION

syntax: IF␣[< memory ref > | < register] > | = | < [< memory ref > | < register > | < constant >]

This command may only be used after an AT command. It allows a conditional execution of the command string attached to this breakpoint.
The content of an address or register is compared with the content of another address, register of constant.
If the condition specified is true the command string is executed. If not the user program is restarted (implicit GØ command).
If < memory reference > is an absolute address then < memory reference > must be specified as M␣ < hexa number >.

```
D:AT $SYMB&FOL4

D:CO /2

D:DM /651A,/6526

D:GO

D:AT $SYMB&FOL5

D:DM /651A,/6526

D:RT

D://
1688
BP: 6CE4
NUMBER UNKNOWN
1835

D:RX

S:
```

Example 6.7

TRACE

syntax: TR␣ <2 ASCII char>

This command can be used to check a condition after a branch instruction which may cause the user program to enter a loop.
The two ASCII characters specified are printed out each time the program reads the breakpoint to which this command is attached.
Restriction:
The ASCII characters may not be
— a space
— a full stop

```
YOUR LOAD ADDRESS = 6504

D:AT /6CA8

D:TR LO

D:GO

D://
BP: 6CA8
LO
BP: 6CA8
LO
BP: 6CA8
LO
BP: 6CA8
```

Example 6.8

READ FROM EXTERNAL DEVICE

syntax: RE␣/ <file code>,<memory reference>,/ <no of characters>

Through this command a number of characters may be read in a buffer whose first address is <memory reference>. The data is read from the device with the specified file code.
When this command is given a standard read is sent to the monitor with the specified address and number of characters.
The number of characters must be specified hexadecimally.
The message READ is printed on the typewriter log when the user requests a record to be read from the typewriter.

START USER PROGRAM

syntax: //

This command must be used to start the execution of the user program after having defined one or more breakpoints. The user program runs until a breakpoint is encountered and commands defined at that breakpoint are then executed by the Debugging Package.
This command has the same function as GØ <start address> when the latter command is used on-line.

EXIT

syntax: RX

This command causes an exit from the Debugging Package and switches control back to the Control Command Interpreter which types out S:.

The following error messages are output when an illegal condition occurs.

MESSAGE	MEANING
UNKNOWN BP	— the Package is asked to delete a breakpoint which has already been deleted — the DB command contains an incorrect address.
BP DOUBLE DEFINED	— the breakpoint with the specified address already exists in the B P table.
REFUSED IN ON-LINE MODE	— the IF command is given not immediately following an AT command.
REFUSED IN OFF-LINE MODE	— an attempt is made to define a new breakpoint without having terminated the previous one by a GØ or RT command.
BP CANNOT BE DELETED	— the current breakpoint cannot be deleted.
BP TABLE OVERFLOW	— the BP table may not contain more than 8 breakpoints. Delete from the table some breakpoints already executed or considered no longer necessary.
NO BP ON LKM/MLK	— the breakpoint may not point to a LKM or MLK instruction.
PARAMETER ERROR	— this message is printed when an illegal parameter is specified.
SYNTAX ERROR	— the syntax in the command is erroneous.

MESSAGE	MEANING
FILE CODE NOT ASSIGNED	— the file code specified was not assigned before the Debugging Package was called.
COMMAND UNKNOWN	— the command given does not belong to the list of commands discussed in the previous chapter.
SYMBOLIC REF. ERROR	— the symbolic reference given does not exist in symbol table.
NO START ADDRESS	— start address missing in the module to be debugged
COMMAND TABLE OVERFLOW	— not enough room to record the command string in the command table
INVALID ADDRESS	— relative address not within program limits

PART 7 ROM IMAGE GENERATOR

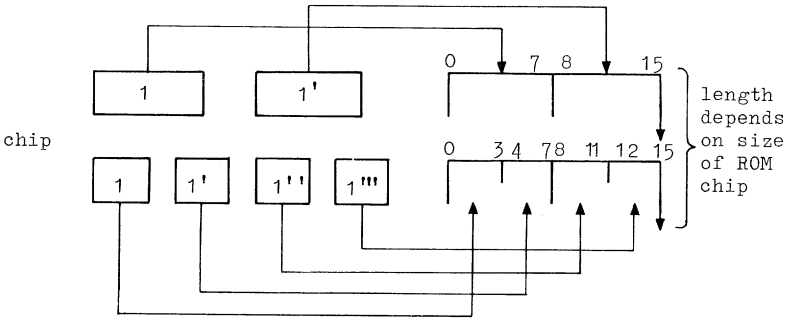
ROMIMAGE is a program by means of which punched tapes are generated containing the memory image of a program to be stored in Read Only Memory chips.

ROMIMAGE may be run under control of the Disc Operating Monitor or as a Stand Alone program (the latter is described in P800M Programmer's Guide 1, Volume VII).

The produced punched tape is used as input for the Data I/O device which reads it and writes a copy of it in the ROM Chip(s).

The minimum configuration required for execution of ROMIMAGE under DOS is:
- CPU + 16k words of memory
- 1 disc unit
- 1 operator's console
- 1 paper tape punch
- optionally, 1 tape reader
- optionally, 1 line printer

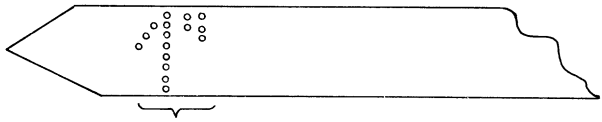
As a basic rule, it can be stated that the location of the ROM chips on the PC board is directly related to their memory layout as represented in the software:



On the tape, each chip is identified by a number:
— for 8-bit chips: x and x′ respectively
— for 4-bit chips: x, x′, x″ and x‴ respectively.

For each chip a punched tape with memory image is produced, after the lower and upper boundaries of the related ROM secton in memory have been specified.

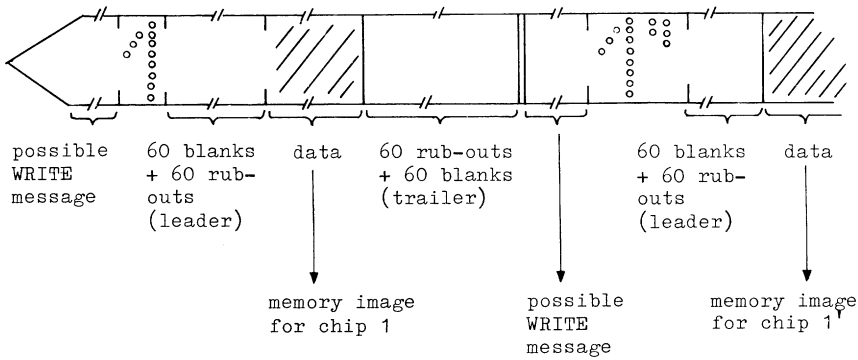
Tape Format
To facilitate identification of the punched tape and the ROM chip it relates to, ROMIMAGE punches an identifier (the chip-number) at the beginning each tape, separated from the data blocks by 60 blank characters and 60 rub-out characters. This identifier is not punched as a coded character, but as a character image, so that it can be easily recognized by the user, though not by the I/O driver:



This identification (here for chip 1′) is always punched automatically. If the user wishes to add personal identification, he can do so by means of a WRITE command.

The length of a data block depends on the size of the chip (at least 512 words). The format of the code as punched in the tape is described below.

A tape generated by ROMIMAGE will look as follows:



Code Format
The memory image can be punched in ASCII BPNF, ASCII hexadecimal or binary code format.

— *BPNF code* example for 8-bit, 512-word ROM chip:

<leader> : 60 blanks, followed by 60 rub-out characters

<SOH character>

BPPNNNNNPF : character field 0

<space>

BPNPNPPPPF : character field 1

<space>
|
etc.
|
BNPNPNNNNF : character field 511

<space>

<trailer> : 60 rub-out characters, followed by 60 blanks

For 4-bit ROM chips, only 4 bits are coded between B and F

— *Hexa code* example for 8-bit, 512-word ROM chip:

```
<leader>                  : 60 blanks, followed by 60 rub-out characters
<SOH character>
<hexa digit>  }
<hexa digit>  }           character field 0
<space>
<hexa digit>  }
<hexa digit>  }           character field 1
<space>
   |
 etc.
   |
<hexa digit>  }
<hexa digit>  }           character field 511
<space>
<trailer>                 : 60 rub-out characters, followed by 60 blanks
```

For 4-bit ROM chips, there is only one hexa digit between spaces.

Binary code example for 8-bit, 1024-word ROM-chip:

```
<leader>                  : 60 blanks, followed by 60 rub-out characters
</FF>                     : start code
<8-bit binary code>       : character field 0
<8-bit binary code>       : character field 1
   |
 etc,
   |
<8-bit binary code>       : character field 1023
<trailer>                 : 60 rub-out characters, followed by 60 blanks
```

For 4-bit ROM-chips, only 4-bit codes are punched after start code/FF.

The following rules must be taken into account:

- there will be exactly as many characters in consecutive sequence on the tape, as the number indicated by the chip size.
- a leader of 60 blanks + 60 rub-out characters precedes the start code (SOH in the 2ASCII formats,/FF in binary).
- a trailer of 60 rub-out characters followed by 60 blanks must follow the last space in the ASCII formats, the last coding in the binary format
- each character field in BPNF code consists of 10 consecutive characters for 8-bit ROM-chips and 6 characters for 4-bit ROM-chip, the first of which must be the start character B.
  Following the start character B, there are exactly 8 or 4 data characters (Ps or Ns). The final character will be F. No other characters are allowed anywhere in the character field.
  Within the character field, P stands for Binary 1
  N stands for Binary 0.
- character fields are separated by a space in the ASCII formats.
- the last field on the tape is followed by a space (binary format excepted) and the trailer. No comments will be punched in between.

ROMIMAGE runs under control of the Disc Operating Monitor.
It can be catalogued as a permanent load module in the user library or it can be made part of the System Disc pack. In both cases it is started by means of the CCI Command RUN.

File Codes
With ROMIMAGE the following file codes must be used and assigned (by CCI command ASG) by the user if they are not implicitly assigned in his system:

- *input command file code:* /0A
  from this file code ROMIMAGE reads all input commands and some control data required for generation of the memory image on punched tape.
- *output command file code:* /0B
  on this file code the ROMIMAGE commands are output.
- *input file code:* /D6
  this file code corresponds to the /L file code, i.e. the Disk Linkage Editor output file code. This is one of the standard file codes, which needs not be assigned by the user.
- *output file code:* /03
  through this file code ROMIMAGE outputs the memory image of the load module, read on the input file, onto punched tape. This file code must be assigned to a paper tape punch.
- *list file code:* /02
  on this file code ROMIMAGE lists all control messages and data which are interchanged between ROMIMAGE and the operator. If this code is assigned to the same device as the output commande file code, no listing will be produced.
- *check file code:* E1
  from this file code ROMIMAGE reads the punched tape containing the memory image of the ROM-chips (one at a time), to verify that it is identical with the memory image on the input file /L.

When ROMIMAGE is started it outputs a sequence of commands for each of which the user must enter the parameters to enable ROMIMAGE to generate the correct memory image for one or more ROM Chips.
ROMIMAGE outputs the command, for example as follows:
ROM: FORMAT OF CODE
then proceeds to the next line
on which the user must type in the parameters, followed by

(LF) (CR)

Below, the commands and the required parameters are described.

FORMAT OF CODE
When this message is output, the user must specify the format in which the memory image is to be written.

ROM: FORMAT OF CODE

<format>

where

<format> = BP[NF]|HE[XA]|BI[NARY][,I]

If BP or BPNF is specified, the memory image will be written in BPNF format (see "General Principles")
If HE or HEXA is specified, the memory image will be written in hexadecimal (see "General Principles")
If BI or BINARY is specified, the memory image will be written in binary (see "General Principles")
,I must be specified if the chips onto which the memory image must be written are of the type VOL (Voltage Output Low), and the DATA I/O device used does not provide the inversion toggle switches for the different input formats; in this case the data on the Input tape must already have been inverted. Thus, with the I option specified, ROMIMAGE inverts the Ps and Ns in the BPNF format and the 0s and 1s in the binary format, whereas in hexadecimal format 0 becomes F, 1 becomes E, etc.
Default value: HEXA

DIMENSION OF CHIP
When this message is output the user must specify the size of the ROM chip for which the memory image will be written.

ROM: DIMENSION OF CHIP

<size>[,<nbr of bits>]

where

<size> is specified as 0 when the chip size is 512 words
1 when the chip size is 1024 words
2 when the chip size is 2048 words
<nbr of bits> indicates the size of a word on the chip, i.e. either 4 or 8, where the default value is 8.

LOADING ADDRESS

When the message is output the user must specify the address at which the load module will be loaded into memory.

Note: This address differs from the lower boundary address specifying the start of that part of the module which is to be located in ROM. The loading address must be smaller then the lower boundary address.

ROM: LOADING ADDRESS

<address>

where

<address> is a value of 4 hexadecimal digits specifying the memory location where the module will be loaded.

BOUNDARIES

When this message is output, the user must specify two addresses indicating the first location of the ROM in which the memory image is to be generated and the first free location following it.

Note: See Note under LOADING ADDRESS. Moreover, it must be remembered that the ROM-bootstrap which is used to start execution of programs in ROM, supposes the first executable instruction to be located at the first word of the ROM-block, i.e. at the address specified as lower boundary.

ROM: BOUNDARIES

<lower boundary>,<upper boundary>

where

<lower boundary> and <upper boundary> must be specified as values of 4 hexadecimal digits, the former value being smaller than the latter.

WRITE MESSAGE?
Before it punches the memory image on the paper tape, ROMIMAGE allows the user to have his own message punched on the tape. This message is punched as an image of the character specified on the command input file. This must be repeated for each chip separately.

ROM: WRITE MESSAGE?

(CR) |<message> (CR)

where

(CR) is typed in if no message is to be punched.
<message> is the message of which the image will be punched on the tape before the chip identifier. Its maximum length is 12 characters.
Only the digits 0 through 9 and the letters A, B and C may be used.

CHECK?
When ROMIMAGE has completed punching the memory image of a ROM chip, it outputs this message to ask the user if he wants the output to be verified.

ROM: CHECK

YE[S]|NO

where

YE[S] is specified if verification is required. The produced punched tape must be put on the tape reader before the command is typed in.
ROMIMAGE reads the tape and compares the input with the memory part of that module.
NO is specified when no verification is wanted. After asking WRITE MESSAGE?, ROMIMAGE continues to the punching of the next ROM chip's memory image.

When ROMIMAGE has verified the tape and found that it is correct, it prints the message:

ROM: CHECK CORRECT

and continues.
When an error has been encountered during the verification process, it prints the message:

ROM: ERROR DURING CHECK

followed by

ROM: CHECK AGAIN?

If the reply is YES, the program immediately starts re-reading the tape, which is assumed to have been put back to its starting position, for another check.
If the reply is NO, the program types out:

ROM: PUNCH AGAIN?

In reply to this message the user can type YES to have the process for the last ROM chip restarted, or NO in which case the process for the following ROM chip, if any, is started.

END OR NEXT BOUNDARY
This message is output when the memory image within the specified boundaries has been punched.
The user is given the possibility to either specify the next boundaries for the same load module or end the ROMIMAGE process.

ROM: END OR NEXT BOUNDARIES

EN[D]|NE[XT]

When EN or END is typed in, the process is terminated.
When NE or NEXT is typed in, other boundaries of the same modules must be punched. ROMIMAGE prints the message.

ROM: BOUNDARIES

Error Messages

The following error messages may be output by ROMIMAGE

- ROM: ERROR OVERLAY LM
  When it has printed this message, ROMIMAGE exits after printing the END message, if possible.
- ROM: NOT ENOUGH CORE
  When it has printed this message, ROMIMAGE exits after printing the END message, if possible.
- ROM: SYNTAX ERROR
  When it has printed this message, ROMIMAGE repeats the last command, to give the user the possibility of a retry.
- ROM: INCORRECT BOUNDARIES
  When it has printed this message, ROMIMAGE repeats the last command, to give the user the possibility of a correct retry.
- ROM: IRRECOVERABLE ERROR ON FILE XX STATUS XXXX
  where X is a hexadecimal character.
  When it has printed this message, ROMIMAGE exits after printing the END message.
- ROM: ERROR DURING CHECK
  this message is printed when an error is detected during the verification process initiated by the control command ROM: CHECK. It is followed by the question ROM: PUNCH AGAIN, to which the user can reply YES or NO. See under command ROM: CHECK.

PART 8 UTILITY PROGRAMS

The P800M computes permits the user to load and run an Initial Program Loader by simply pressing the IPL button on the front panel. When doing so the contents of a 64-word Read Only Memory (ROM). which has been preprogrammed with a standard bootstrap, is read into the CPU's memory where it is-executed.
Next the IPL, which is punched in front of the system program or which may be an independent tape or may be written on cassette tape, magnetic tape or on disc, is automatically loaded and run from one of the following devices:

- operator's typewriter (4 × 4)
- punched tape reader
- fixed head disc
- moving head disc
- magnetic tape
- cassette tape.

However, before the button is pressed the user must set the following parameters on the data switches:

bit 0 = 1 IPL is to be loaded from ASR (4 × 4)
0 IPL is to be loaded from other device
bit 1 = 1 IPL from disc
0 IPL from other decive
bit 2 = 1 moving head disc
0 fixed head disc
bit 3 = 1 Programmed Channel transfer
0 I/O Processor transfer
bit 4 = 7 control information for control unit during execution of CIO Start sent by the bootstrap. Specify the type of device:
TY = 0001 X1215 =0011 (phys. sect address)
TK = 0111
MT = 0010

bit 8 = 1 multiple device control unit
0 single device control unit
bit 9 = 1 X1215 disc (P824-001 CDD disc)
10 – 15 device address of device from which the IPL is loaded

When the IPL is loaded from disc the processing of bootstrap and IPL takes place as follows:

- the bootstrap loads sector 1 (physical sector 3) from disc and copies it in location /80 onwards.
- a part of the IPL receives control at location /84 and:
  - computes the memory size
  - shifts the remaining part of the IPL into the top of memory and gives control to it.
- this part of the IPL loads the Supervisor, located in sector 12 and higher, from disc and gives control to the initialization part of the monitor (INIMØN).

Loading procedure

- prepare device from which the IPL is read
- set the relevant information on the data switches
- push the IPL button. The IPL is now read.

The ASCII dump program allows to have an ASCII memory dump in hexadecimal format on the line printer or on the operator's typewriter. The program is delivered on paper tape in 8+8 or 4×4 format and is preceded by an IPL.
The IPL is loaded by pushing the IPL button on the control panel. The user may now specify in register A9 the loading address of the dump program. Default value = 0000.
Next push the RUN button to load the dump program. When the reading stops to the user must load register A8, A9 and A10 with the following information:

A8 to be loaded with the address of the device onto which the dump will take place, e.g./10 for the operator's typewriter or /07 for the line printer.
If the device is connected via the programmed channel bit 0 of register A8 must be set to 1.
A9 to be loaded with the first address of the area to be dumped
A10 to be loaded with the last addres of this area.

Press the RUN button to activate the dumping.

APPENDICES

01	Operator's typewriter
02	Print Unit
03	Punch Unit
04 - 09	Reserved
D0	Catalogued procedure input
D4	/S file or library source file (Line Editor output)
D5	/O file (Assembler output, Linkage Editor input)
D6	/L file (Linkage Editor output)
D7	System object file (library)
D8	User object file (library)
D9 - DF	Reserved for system use
E0	Control Command Input
E1	Source Input
E2	Object Input
EE	Catalogued procedure output
EF	System operator's typewriter
F0 - FF	Logical addresses of disc unit. They are reserved for system use.

Note: 0A - CF may be used after an assignment
01, 02, 03, E0, E1, E2, EF may be used without having been assigned in a previous ASG control command.

The file codes 04 to CF may be used to address user files.
File codes 0A to DF are scratched after SCR of BYE.

Appendix B Non-ASCII format on paper tape

The information on paper tape may be in object format

Object format
Object format information is organized in logical records called clusters. This is the format as output by the Assembler. Linkage Editor and Compilers. Object format also is the input for the Linkage Editor, IPL and Basic Monitor Loader. 8+8 format.

Punched format of paper tape
The punched format of information on paper tape determines from which device the punched tape may be read. The formats are:
- 8+8 format
- 4×4 format.

8+8 format
Punched tape of this format can only be read from the high speed paper tape reader and not from the ASR paper tape reader.
Each hole of the 8-hole channel represents one character bit, the rightmost one being on the sprocket row side.

4×4 format
Paper tape punched in this format may be read from either the high speed paper tape reader or the ASR tape reader. One 8-bit character requires two rows four holes. Each row represents four bits. The first row corresponds to the leftmost 4 bits of the character.

Tape representation	4 bits value
10	0
1 to 5	1 to 4
15 to 1F	5 to F

Standard paper tape device handlers always accept either 8+8 or 4×4 format

Appendix C Object code

The object format described below is the output, on paper tape, of the Disc Assembler and of the FORTRAN compiler.
The object code is provided in two formats:

- 8 + 8 format where the information, contained in 16-bit words, is punched over two rows of 8 bits. This format can be read from the high speed punched tape reader.
  The first character of each record is a record identification (X'10, X'1' to X'4', X'15' to X'17').
  The second character specifies the number of 16 bit words in the remainder of the record, the checksum excluded.
  The remainder consists of 16-bit words punched over two rows. The last word of each record is the checksum.
- 4×4 format where the information, contained in 16-bits words is punched over four rows of four bits. This format can be read from the high speed punched tape reader and from the reader attached to the operator's typewriter. The first character of each record is a record identification X'18' to X '1F').
  The second character specifies the number of 16-bits words in the remainder of the record, the checksum excluded.
  The remainder of the record consists of n 16-bit words.
  The last word of the record is a checksum.

The characters in object code records are modified in the following way: add X'10' to zero characters and to binary characters X'5' to X'7'. Other characters remain unchanged.

Cluster type	8+8 format	4×4 format
0	/10	+8 =/18
1	1	+8+/10=/19
2	2	+8+/10=/1A
3	3	+8+/10=/1B
4	4	+8+/10=/1C
5	/15	+8 =/1D
6	/16	+8 =/1E
7	/17	+8 =/1F

If the first character is ≠/07 (Bell),/0A (Line Feed), /0D (CR),/11 (Xon), /13 (Tape Off), /7F (Rub Out), /12 (Xon punch), /14 (Xoff punch) and <20 then it is object code.
Only the last four bits are kept (∧ /F). If the value is <8 then the object code is of format 8+8 and if >8 then 4×4. To find the cluster type 8 must be subtracted.

In P800M Object Code there are 8 types of cluster, namely

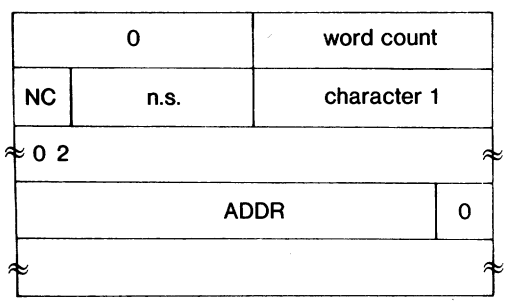
– BLOCK DATA CLUSTERS	type 0
– ENTRY POINT NAMES CLUSTER	type 1
– EXTERNAL REFERENCE NAME CLUSTER	type 2
– CODE CLUSTER	type 3
– INTERNAL MODIFICATION CLUSTER	type 4
– ENTRY POINT DEFINITION CLUSTER	type 5
– COMMON LENGTH DEFINITION CLUSTER	type 6
– END CLUSTER	type 7

The first record of any object module is a program identification record is ASCII format and consists of an IDENT statement which is the same as the source statement from which it originated, followed by LFXOFF CR.

The eight types of clusters do not have the same length.

BLOCK DATA CLUSTER (0)

This cluster is only generated by the FORTRAN compiler when a Block Data subroutine is processed. It permits the initialization of a COMMON block.



name of labelled COMMON block (up to 6 characters)

absolute data words to be loaded, starting at ADDR in named COMMON block

Max. 34 words are allowed.

where:

- the first character of the first word indicates the type number of the cluster (0).
- the second character of the first word contains the number of words in the remainder of the cluster, excluding the checksum.
- bit 0, 1, and 2 of the second word contains the number of characters (max 6) specifying the name of the labeled COMMON block. Bit 3 to 7 are not relevant.
- ADDR, bit 15 = 0 indicates the relative address of the first data word to be loaded in the named COMMON block.

ENTRY POINT NAMES CLUSTER (1)

This cluster contains a list of entry point names. Entries in this cluster have no fixed length.
All clusters of this type are grouped and follow the IDENT.
Each entry point name is given a value in the ENTRY POINT DEFINITION CLUSTER.

1	word count
NC (=4) n.s.	N
0 2 3 7 A	M
0 E	0 0
NC (≠1) n.s.	E
NC (=3) n.s.	E
0 2 3 7 N	T

Max. 34 words

where:

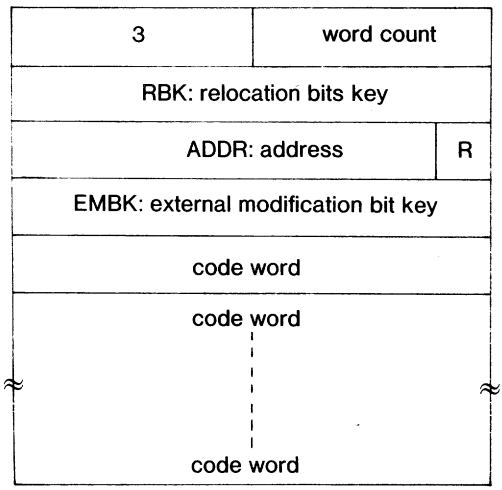
- the first character of the first word indicates the type number of this cluster
- the second character of the first word indicates the number of words in this cluster
- NC gives the number of characters in the name
  A maximum of six characters is allowed
- n.s. = not significant.

EXTERNAL REFERENCE NAME CLUSTER (2)

This cluster contains a list of external reference names. These names are referred to in the code cluster by a number; the first external reference name encountered in the module has received number 2, the second one number 4 etc. In the remainder of the object program the externals may be referred to by their number.
The format of this cluster is the same as for the type 1 cluster.

CODE CLUSTER (3)

This cluster contains the code words (e.g. instructions) generated by the Assembler and Linkage Editor and the addresses from which the code words have to be loaded into memory. Each cluster may contain a maximum of 16 code words.



Max. 34 words

where:

- the first character of the first word indicates the type number of this cluster.
- the second character of the first wordt indicates the number of words in this cluster.
- RBK is a Relocation Bit Key. Bit 0 of this word is related to the first code word in this cluster, bit 1 to the second word etc.
  If a **code word** has its RBK bit set (to 1) this code word is relocatable. At link edit time the relative address of the object module will be added to the address of this word
- ADDR contains the address from which the list of code words in this cluster must be loaded into memory.

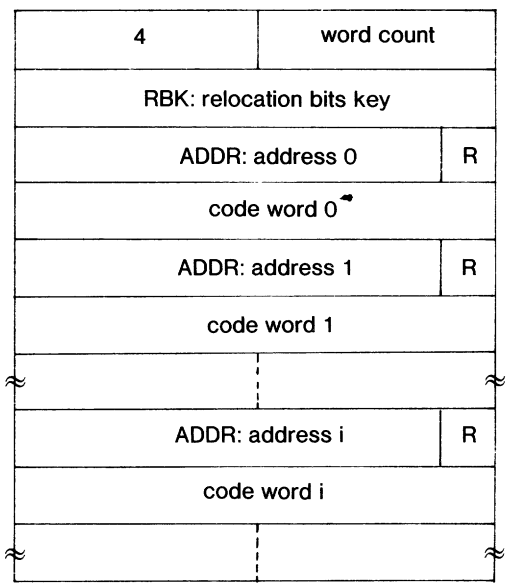
If bit 15 (R) = 0, ADDR is the absolute address of the first code word of this list (absolute program section). In that case the RBK bit is reset except when there is a reference from an absolute program section to a relocatable program section.
If bit 15 = 1, ADDR is the relative address, i.e. the address of the first code word of the list relative to the load address of the module (relocatable programm section).

– EMBK is an External Modification Bit Key. As for RBK, bit 0 is related to the first code word, bit 1 to the second etc.
If a code word has its EMBK bit set the next word in the list is not a code word but a word that contains the number of external reference.

The purpose hereof is that the previous code word has to be modified by the Linkage Editor by adding the numerical equivalent of the external reference to the code word.

INTERNAL MODIFICATION CLUSTER (4)

Code words generated by the Assembler and Linkage Editor are listed in this cluster each time a forward reference is satisfied.
Each code word in this cluster is associated with a relocation bit of a RBK as described in cluster type 3, and an absolute or relocatable word address.
Bit 0 of RBK is related to code word 0, bit 1 to code word 1 etc.

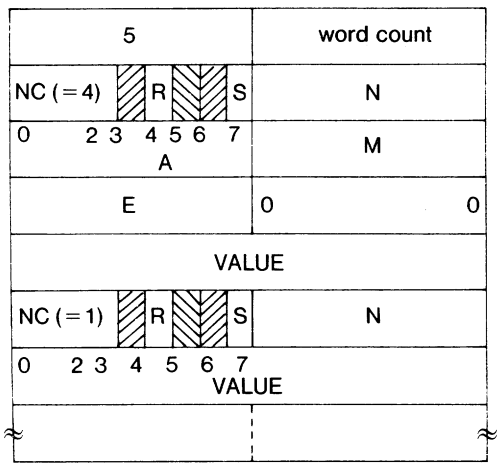


where:

- the first character of the first word indicates the typenumber.
- the second character of the first word indicates the number of words in this cluster, checksum excluded.
- R, if 0 the address is absolute
  if 1 the address is relocatable

ENTRY POINT DEFINITION CLUSTER (5)

This cluster contains a list of entry point names and the number they have been given for reference purposes.



Max. 34 words

where:

- NC is the number of characters in the entry point
- R if 1, the entry point name is relocatable
  if 0, the name is absolute
- S if 1, the name is the name of an internal symbol table
  if 0, if not
- NAME the name of the entry point
- VALUE value of the entry point

COMMON LENGTH DEFINITION CLUSTER (6)

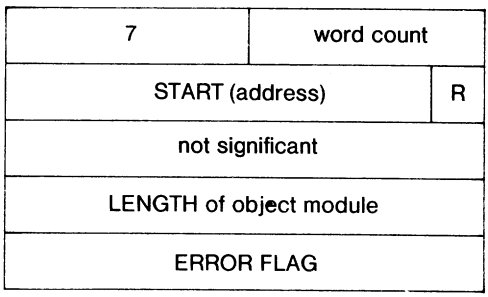
This cluster is output by the Assembler and by the FORTRAN compiler.
Those two can make use of a common block in which memory locations are shared between subprogram and main program variables.
References to a common area are considered to be external references to that common with a displacement value representing a relative address in the common.
Since the Linkage Editor has to know the length of the common blocks the length of all blocks are given in this cluster.

The format of this cluster is the same as for the Entry Point Definition cluster, except that R and S are not significant, and

NAME = the name of the common block. If it is a blank common the name is one blank.
VALUE = length in characters of the common block.

END CLUSTER (7)

This cluster is always the last cluster of a module. Its format is as follows:



where:

- the first character of the first word indicates the typenumber of this cluster
- the second character indicates the number of words in this cluster, checksum excluded.
- START is the start address of the object module. If R (bit 15) = 0 the address is absolute, if 1 it is the relative starting address. If START and R are 0, there is no start address.
- LENGTH contains the length of the object module given as an even number of characters.

– ERFLAG is a flag indicating whether any errors have occurred during the assembly or link-editing of the module.
If so, the number of errors is given in binary. If no errors have occurred this word is 0.

char	*ASCII octal*	*Intern Hexa*	*char. set punch comb.*
space	240	20	on punch
!	241	21	11,8,2
„	242	22	8,7
#	243	23	8,3
$	244	24	11,8,3
%	245	25	0,8,4
&	246	26	12
'	247	27	8,5
(	250	28	12,8,5
)	251	29	11,8,5
*	252	2A	11,8,4
+	253	2B	12,8,6
,	254	2C	0,8,3
—	255	2D	11
.	256	2E	12,8,3
/	257	2F	0,1
0	260	30	0
1	261	31	1
2	262	32	2
3	263	33	3
4	264	34	4
5	265	35	5
6	266	36	6
7	267	37	7
8	270	38	8
9	271	39	9
:	272	3A	8,2
;	273	3B	11,8,6
<	274	3C	12,8,4
=	275	3D	8,6
>	276	3E	0,8,6
?	277	3F	0,8,7
	300	40	8,4
A	301	41	12,1
B	302	42	12,2
C	303	43	12,3

char.	*ASCII octal*	*Intern Hexa*	*char. set punchcomb.*
D	304	44	12,4
E	305	45	12,5
F	306	46	12,6
G	307	47	12,7
H	310	48	12,8
I	311	49	12,9
J	312	4A	11,1
K	313	4B	11,2
L	314	4C	11,3
M	315	4D	11,4
N	316	4E	11,5
O	317	4F	11,6
P	320	50	11,7
Q	321	51	11,8
R	322	52	11,9
S	323	53	0,2
T	324	54	0,3
U	325	55	0,4
V	326	56	0,5
W	327	57	0,6
X	330	58	0,7
Y	331	59	0,8
Z	332	5A	0,9
[	333	5B	
\	334	5C	
]	335	5D	
↑	336	5E	
←	337	5F	

Bell	207	07	
Linefeed	212	0A	
Car Ret.	215	0D	
X on reader	221	11	
X off reader	223	13	
Rubout	377	7F	
X on punch	222	12	
X off punch	224	14	
FF		0C	

A

Address expression 1-11
Addressing 1-14
AORG 1-35
Area reservation 1-39
Assembler. 2-1
Assembly control. 1-32
Assembly Language. 1-1
Assembly listing 2-9
Asterisk. 1-11

B

Blank common 1-27
Block data cluster C-2
Branch instructions. 1-18
Breakpoint 6-8

C

Character constant 1-13
Character string 1-13
Cluster C-1
Cluster type C-1
Code cluster. C-4
Condition field. 1-9
Condition indicator. 1-9
Commons. 1-30
Common length definition . . . C-7
Comment field. 1-13
COMN. 1-30
Constant 1-12

D

DATA 1-36
Debugging Package 6-1
Decimal constant. 1-12
Direct addressing 1-14
Directives. 1-23

E

EJECT 1-40
END. 1-26
End cluster C-7
ENTRY 1-28
Entry point names cluster . . . C-3
Entry point definition cluster. . C-6
EQU 1-38
Error messages. 2-11, 3-10
Expression 1-10
External reference 1-29
External reference
names cluster C-4
EXTRN 1-29

F

File codes. A-3
FORM 1-41
Format of source statements . . 1-7

G

GEN. 1-46
Glossary of terms IX

H

Hexadecimal constant. 1-13

I

IDENT. 1-25
Identifier 1-7
IFF 1-33
IFT 1-33
Indexed addressing 1-14
Indexed indirect addressing . . 1-15
Indirect addressing 1-15
I/O instructions 1-21
Internal modification cluster . . C-5
Internal symbol table 1-34

L
Label 1-7
Labeled common. 1-27
Line Editor 5-1
Linkage Editor 3-1
Linkage control 1-27
LIST 1-40
Listing control. 1-40
Load module 3-9
Location counter. 1-4

M
Machine instruction 1-3
Map 3-11
MMU 1-54

N
NLIST 1-40

O
Object code C-1
Object module. 3-5
Object module library 3-5
Operand field 1-10
Overlay linkage editor 4-1

P
Page fault interrupt. 1-55
P-register 1-12
Predefined expression 1-11
Programming considerations. . 1-49
Program framework 1-24
PSW 1-50

R
Register addressing 1-15
Register expression 1-12
Register to register operation. . 1-15
RES 1-39
ROMIMAGE Generator . . . 7-/
RORG 1-35

S
Segment table 1-55
Shift instructions 1-20
Simulation 1-56
Source module. 1-7
STAB 1-34
Statement 1-7
Store indicator. 1-9
Symbol 1-7
Symbol table 1-34
Symbol generation 1-41

T
Trap 1-54

U
User stack 1-50

V
Value definition 1-36

X
XIF 1-33
XFORM 1-45

Notes

Comment sheet

P800M PROGRAMMER'S GUIDE 2 - Volume III: Software Processors

Pub. No. 5122 991 27392

Name: _____

Company: _____

Department: _____

Address: _____

_____

Telephone number: _____ Ext. _____

Comments or Suggestions:

PHILIPS DATA SYSTEMS B.V.

MARKETING GROUP SMALL COMPUTERS
P.O. Box 245, Apeldoorn, The Netherlands
Phone: 055-230123; telex: 49142

For further details contact the above address or:

Sweden
Svenska AB Philips
Data Systems
Minidatorer
Rissneleden 16
Stockholm 27
Tel. 08 830300

France
Philips Data Systems
Département Mini-ordinateurs
5 Square Max-Hymans
75015 Paris 15
Tel. 01 734 7759

Switzerland
Philips AG
Data Systems
Binzstrasse 18
8027 Zürich
Tel. 01 442211

Norway
Norsk A/S Philips
Data Systems Division
Nils Hansens vei 2
Oslo 6
Tel. 02 679380

Western Germany
Philips Data Systems GmbH
Weideneuerstrasse 211/213
Siegen-Weidenau 21
Tel. 027 14041

Great Britain
Philips Data Systems
Elektra House
2 Bergholt Road
Colchester C04-5AA
Essex
Tel. 206 5115

Belgium
Philips Data Systems SA
Marketing Group Small Computers
Anspachlaan 1
1000 Brussel
Tel. 02 2193900

Austria
Philips Data Systems GmbH
Untere Donaustrasse 11
A1020 - Wien 2
Tel. 0222 2475610

The Netherlands
Philips Data Systems Nederland B.V.
Bordewijkstraat 4
Rijswijk (Z-H)
Tel. 070 906720