

PHILIPS

PTS 6800 TERMINAL SYSTEM

User Library

PTS 6800 ASSEMBLER PROGRAMMER'S REFERENCE MANUAL

Part 3

Module M06



**Data
Systems**

Date : January 1978

Copyright : Philips Data Systems B.V.
Apeldoorn, The Netherlands

Code : 5122 993 42131

PREFACE

The Assembler Programmer's Reference Manual provides the information required to write, process and test Assembler application programs for the PTS 6800 computer used in the Philips PTS 6000 Terminal System.

Information is divided into three parts as follows :

Part 1 : Assembler Language
Additional functions
Recommended techniques

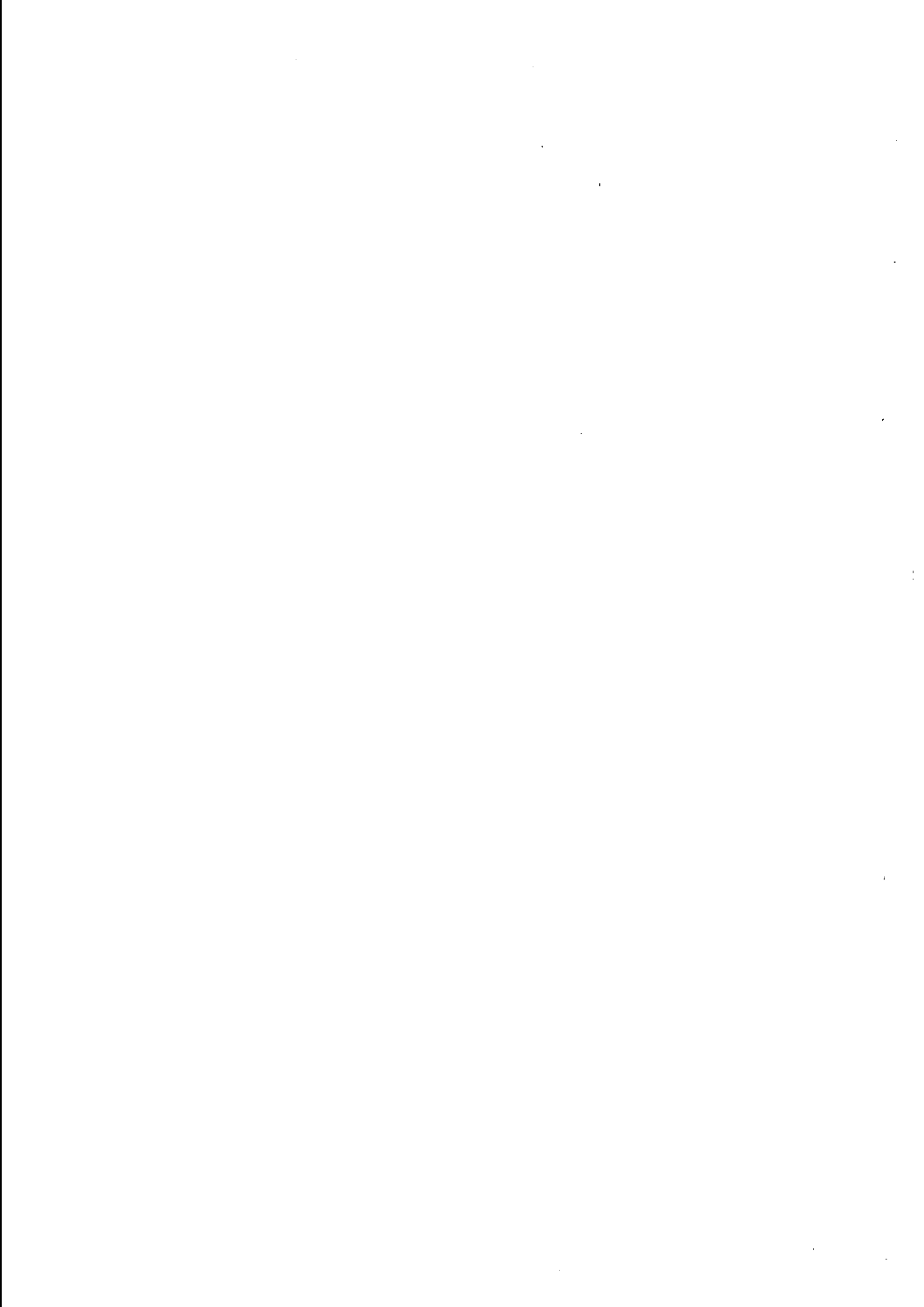
Part 2 : Monitor requests
I/O drivers
TOSS utilities

Part 3 : Assembler processor
TOSS system start
Assembler debugging program

Parts 1 and 2 contain the information needed to write an Assembler program. Part 3 contains the information needed to process and test an Assembler program.

Processing of Assembler programs (updating, assembling, etc.) is done under DOS 6800 System Software. The use of those parts of DOS 6800 System Software designed specifically for Assembler programs is described in Part 3 of this Manual. Information concerning the use of the general purpose components of DOS 6800 System Software is contained in the DOS 6800 System Software PRM (M11). Readers of the present Manual are expected to be familiar with the contents of the DOS 6800 System Software PRM.

The testing and production running of Assembler application programs is done under TOSS System Software. Information concerning TOSS System Software which is relevant to the writing, testing and running of Assembler application programs is included in Part 2 of the present Manual.



CONTENTS

	Date	Page
PREFACE	Jan. 1978	0.0.0
1. INTRODUCTION	Jan. 1978	1.0.1
	Jan. 1978	1.0.2
2. ASSEMBLER PROCESSOR		
2.1. General	Jan. 1978	2.1.1
	Jan. 1978	2.1.2
2.2. Input	Jan. 1978	2.2.1
	Jan. 1978	2.2.2
2.3. Output	Jan. 1978	2.3.1
	Jan. 1978	2.3.2
	Jan. 1978	2.3.3
	Jan. 1978	2.3.4
	Jan. 1978	2.3.5
3. TOSS SYSTEM START		
3.1. General	Jan. 1978	3.1.1
3.2. System Start Procedure	Jan. 1978	3.2.1
	Jan. 1978	3.2.2
	Jan. 1978	3.2.3
3.3. Deferred binding of Monitor Configuration Data	Jan. 1978	3.3.1
	Jan. 1978	3.3.2
	Jan. 1978	3.3.3
	Jan. 1978	3.3.4
4. ASSEMBLER DEBUGGING PROGRAM		
4.1. Introduction	Jan. 1978	4.1.1
4.2. Using DEBUG	Jan. 1978	4.2.1
4.3. DEBUG Input	Jan. 1978	4.3.1
	Jan. 1978	4.3.2
	Jan. 1978	4.3.3
	Jan. 1978	4.3.4
	Jan. 1978	4.3.5
	Jan. 1978	4.3.6
	Jan. 1978	4.3.7
	Jan. 1978	4.3.8
	Jan. 1978	4.3.9
4.4. Running DEBUG	Jan. 1978	4.4.1
	Jan. 1978	4.4.2
APPENDIX A : MEMORY ORGANIZATION		
A.1. Memory Layout	Jan. 1978	A.1.1
	Jan. 1978	A.1.2
A.2. Interrupt System	Jan. 1978	A.2.1

1. INTRODUCTION

An Assembler program may be input to the PTS 6000 system via any input device. After input, the source module is held on disk.

All processors and utilities mentioned in the flow-diagram read input from disk and write output to disk.

The following diagram illustrates the sequence of processes needed to develop and run an executable program from Assembler source modules.

Each source module is processed separately by the Assembler.

The Assembler produces object code modules. Each module may contain references to :

- Labels in the same module
- Labels in other Assembler modules
- Assembler System routines

These references are satisfied by the Linkage Editor. This processor builds an application load module from the following object modules :

- Assembler application modules
- Assembler system routines (if referenced)

The output result is an application program load module. This module may be stored on magnetic tape cassette, TOSS formatted disk, or flexible disk.

The Assembler and Linkage Editor are run under the DOS 6810 Monitor.

The Load module produced by the Linkage Editor, however, must be run under the TOSS Monitor.

If any application program errors are detected during testing, one or more source modules will have to be corrected. This may be done via the Line Editor — an interactive text editor.

Each corrected source module must be reprocessed by the Assembler, then the whole program must be processed by the Linkage Editor.

The Assembler Debugging Program, if required must be linked to the TOSS Monitor during system generation. The Debugging program is an interactive diagnostic routine which enables the programmer to monitor and control the execution of his program. Control is handed to the Debugging program immediately after the TOSS Monitor and application program are loaded into memory (system start).

Information concerning TOSS System Software which is needed by Assembler programmers is contained in this Manual.

The Assembler processor, though part of DOS 6810 System Software, is discussed in this Manual because it is used by Assembler programmers only.

The remaining DOS 6810 System Software components used by Assembler programmers, notably the Linkage Editor, are described in the DOS 6810 System Software PRM (M11).

2. ASSEMBLER PROCESSOR

2.1. General

An Assembly code program may be input to the PTS 6000 System via an input device. After input the source module is held on disk, either in a library or a temporary source file (/S), according to the programmer's requirements. The Assembler processor is held in the system library. It operates under control of the DOS 6810 Operating System and translates source modules from disk and outputs an object code module to disk. The object module can then be linked with other object modules using the Linkage Editor (see PTS 6000 System Software PRM, M11).

Figure 2.1 shows the sequence of events needed to develop and run an executable program from Assembly source modules.

Each source module is processed separately by the Assembler which produces object code modules. The instructions in these modules use a byte-oriented addressing system. Each module may contain references to :

- Labels in the same module
- Labels in other Assembly modules
- Application modules
- System routines

The Assembler processor optionally gives a listing of the program being processed. This listing provides the programmer with all the information he requires for a full record of the program :

- A line count in decimal
- Location counter in hexadecimal
- Hexadecimal representation of the instruction
- Type of address reference
- Source code statement
- Programmer's comments
- An error code at the place an error occurs
- Table of external labels
- Symbol table
- Total error count

The Assembler DEBUG program can be specified at system generation time if required. DEBUG is an interactive diagnostic task which is executed in parallel with the Assembly program being tested. With DEBUG the programmer can monitor and control the execution of his program.

When errors have been discovered, the programmer can use the Line Editor to update the module with the correct statement (see DOS 6810 System Software PRM, M11).

The Assembler provides a facility for the programmer to insert directives into his program. These directives, although they are not program instructions, allow the programmer to guide the assembly process.

These directives can be inserted in the source program the first time it is assembled or input via the Line Editor. The directives are fully defined in volume 1 of this manual and a list is shown below.

Directive	Meaning
DATA	Data generation
EJECT	Continue listing on new page
END	End of Assembly
ENTRY	Define entry point name
EQU	Equate symbol to value or another symbol
EXTRN	Define external reference
FORM	Format definition
GEN	Generation directive
IDENT	Program identification
IFF	If false
IFT	If true
LIST	Resume listing output
NLIST	Suspend listing output
RES	Reserve memory area
XFORM	Extension of FORM directive
XIF	End of condition

2.2. Input

2.2.1. Preparation of the Source Module

Source modules can be input via any input device.

The statements can be prepared on these media in either of two formats :

1. They can be prepared in the same format as the program coding sheet with tabulation points set at the 1st, 10th, 19th and 41st column.
2. They can be prepared with a space or backslash (\) between each field.

Whatever media the module is being prepared on, the relevant labelling and structure standards apply (see DOS 6810 System Software PRM, M11). The Assembler always lists instructions in coding sheet format.

2.2.2. Source Input

The source module must be input to the system by using the command :

```
RDSL [ /file-code]
```

Where /file-code is the device from which the source module is to be read.

The last statement input must be :EOF, even if the input is from the console typewriter. The source module is placed in a temporary file (/S) on disk.

2.2.3. Example Input

The following coding could be input via any input device :

Data Systems		PTS6810 ASSEMBLER		DATE _____
PROGRAM <u>SOURCE INPUT</u>				PROGRAMMER _____
label	operation	operand	comments	
DATAF	LDK	A4,4		70 72
	ABL(7)	HALT		
DEVUN	LDK	A1,0	SET INDEX REGISTER FOR BUFFER	
	LDK	A3, /FF	LOGICAL CONSTANT IN A3	

This code could also be input as follows :

```
DATAF\LDK\A4,4
\ABL(7)\HALT
DEVUN\LDK\A1,0\SET INDEX-REGISTER FOR BUFFER
\LDK\A3,/FF\LOGICAL CONSTANT IN A3
```

The listing device gives the listing in the same layout as the coding sheet. An example output listing is shown in paragraph 2.3.1.

Note : The IDENT statement cannot be input using the backslash between fields. It must be input with the correct spacing.

2.2.4. Assembly

The Assembler must be called by using the command :

```
ASM □ { /S  
name } [,NL]
```

where :

/S indicates that the source program must be read from the /S file

name indicates the name of a library source module or program to be assembled.

NL if specified, informs the Assembler that no listing is required of the assembled program.

If NL is omitted a listing is produced on the printer.

Error messages, however, will always be listed and the lines on which the errors occurred.

The Control Command Interpreter checks the ASM control command parameters for errors. When there is an error in the parameter, 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. An example sequence could be :

S : ASM

FILE NAME MISSING

S : ASM /S

or

S : ASM < name >

The source module residing in the temporary file is now read and assembled.

Errors in the source program are detected by the Assembler. It is not possible to correct errors during processing. In case of a fatal error during processing e.g. table overflow, core overflow, IDENT missing, END missing, the source module is read until an : EOF mark is encountered. At that moment the following message is printed :

FATAL ERROR HAS OCCURRED NO OBJECT CODE PRODUCED

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

If an I/O error occurs, processing is terminated immediately.

2.3. Output

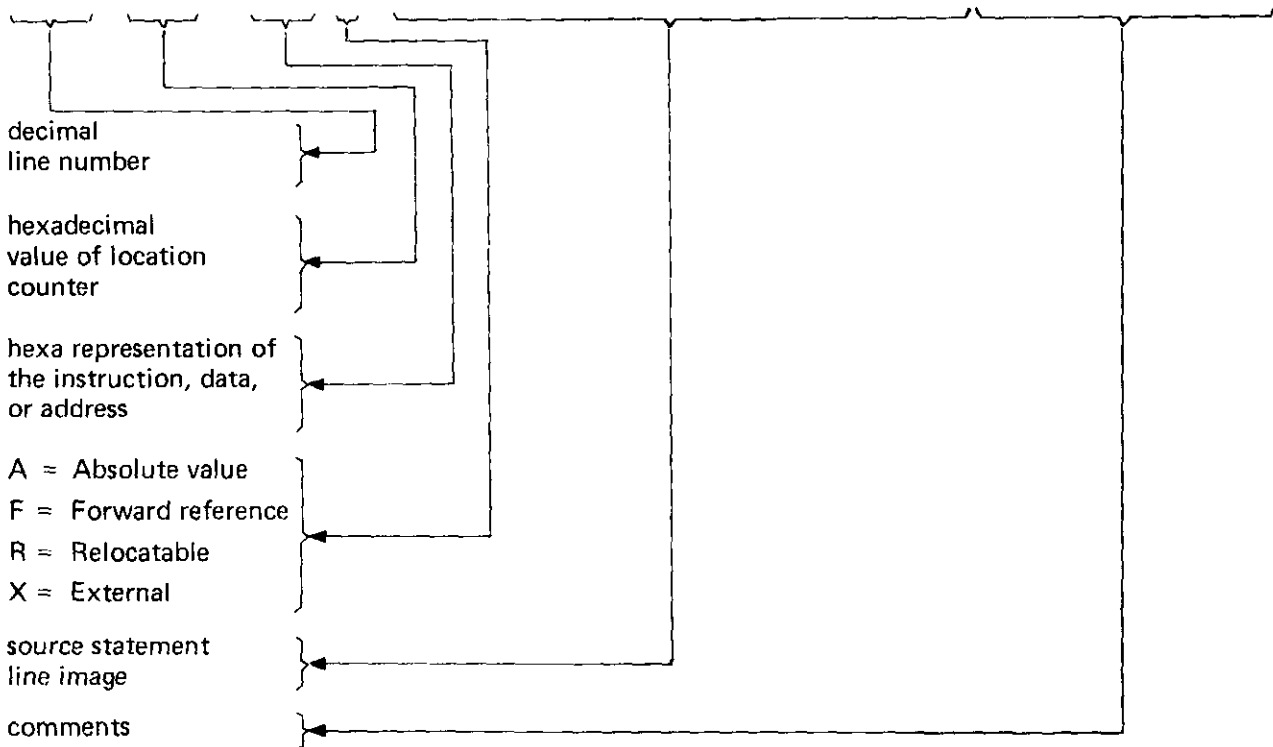
The standard object output file for the Assembler is the temporary/O file. If this file does not yet exist an assignment is made for it. When it does exist the object output is written after the information already existing in this file unless it has been closed by an EOF record. In that case a new/O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

2.3.1. Assembly Listing

The Assembly listing is output on the listing device if the NL option is omitted. The format of the printout is shown in the example below.

```

00000                                IDENT  FORM                                MODULE TITLE
00001                                INOUT  FORM  8 = /07,8,16 = /80A0,16,16 = /2804,16 = 1 SET UP LKM WORDS
00002      0000                      BUFFER RES  10                                RESERVE 10 WORDS BUFFER
00003      0014      0008              DECB  DATA  8,BUFFER,20,0,0,0              CONSTRUCT ECB
                                0016      0000      R
    
```



When a non-fatal error has occurred during assembly the processing continues but the place where the error occurred is indicated by an error code (see Section 2.3.3) following the line in which the error occurred. An asterisk 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 in Symbol Table, below).

2.3.2. Symbol Table

Each label which appeared in the label field of an instruction is given an address relative to the beginning of the module.

The symbol table consists of a list of labels for each name defined in or referred to within the module. This table is always printed out even if the user did not ask for a listing.

The symbol table has the following format :

label value type

where label is the name that appeared in the label field of an instruction,
 value is either the address of the label relative to the beginning of the module or an absolute value (see type below),
 type is the type of label. It can be one of four single digit codes : A, F, R, or X.
 A = an absolute value. The label has been assigned a value with an EQUate directive.
 F = a forward reference. The label is defined later in the module.
 R = a relocatable reference.
 X = an externally defined label (EXTRN)

The symbol table is printed with three labels across the width of the paper. The total number of errors encountered during Assembly is printed at the end of the symbol table. An example is shown below.

```

M:A00    0000 R  M:B00    005C R  MPYMOD                X
ADDMOD           X  DSUMOD           X  SYSAB                X
M:A01    0006 R  T:SOPC    0136 R  M:A02    0018 R
M:A03    0010 R  T:SVR     0152 R  M:A04    0020 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
    
```

2.3.3. Error Messages

The following error messages can be output by the Assembler :

CODE	MEANING	DESCRIPTION
*C	Illegal constant	- "Constant" overflow - A constant with hexadecimal value must begin with either / or X. In the latter case, the value must be enclosed by quote marks, e.g. X 'ZF'. - A constant should not have been written here. - Hexadecimal constant written either X" or /".
*E	Not an even address	- The specified start address is not even. - The specified AORG or RORG operand is not even.
*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.

CODE	MEANING	DESCRIPTION
		<ul style="list-style-type: none"> — 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. — The predefinition of the fields of the current word of an XFORM declaration is 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. — The same field specified twice in the syntax definition line.
*I	Illegal identifier	The first character of a symbol must be a letter.
*L	Illegal label	<ul style="list-style-type: none"> — The label has been defined previously as : <ul style="list-style-type: none"> — a symbol name — an external reference name — an entry point name — A label has been given where it was not allowed. — A label must be specified.
*M	Unknown mnemonic	<ul style="list-style-type: none"> — Unknown mnemonic — Unknown condition mnemonic
*O	Overdisplacement	— Displacement value of parameter too large.
*P	Illegal parameter	<ul style="list-style-type: none"> — 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 must not be an entry point name, a COMMON name or a forward reference. — The operand in a DATA directive must not give more than 16 code words. — " " is not a character string — Illegal use of a register name in a standard instruction operand.

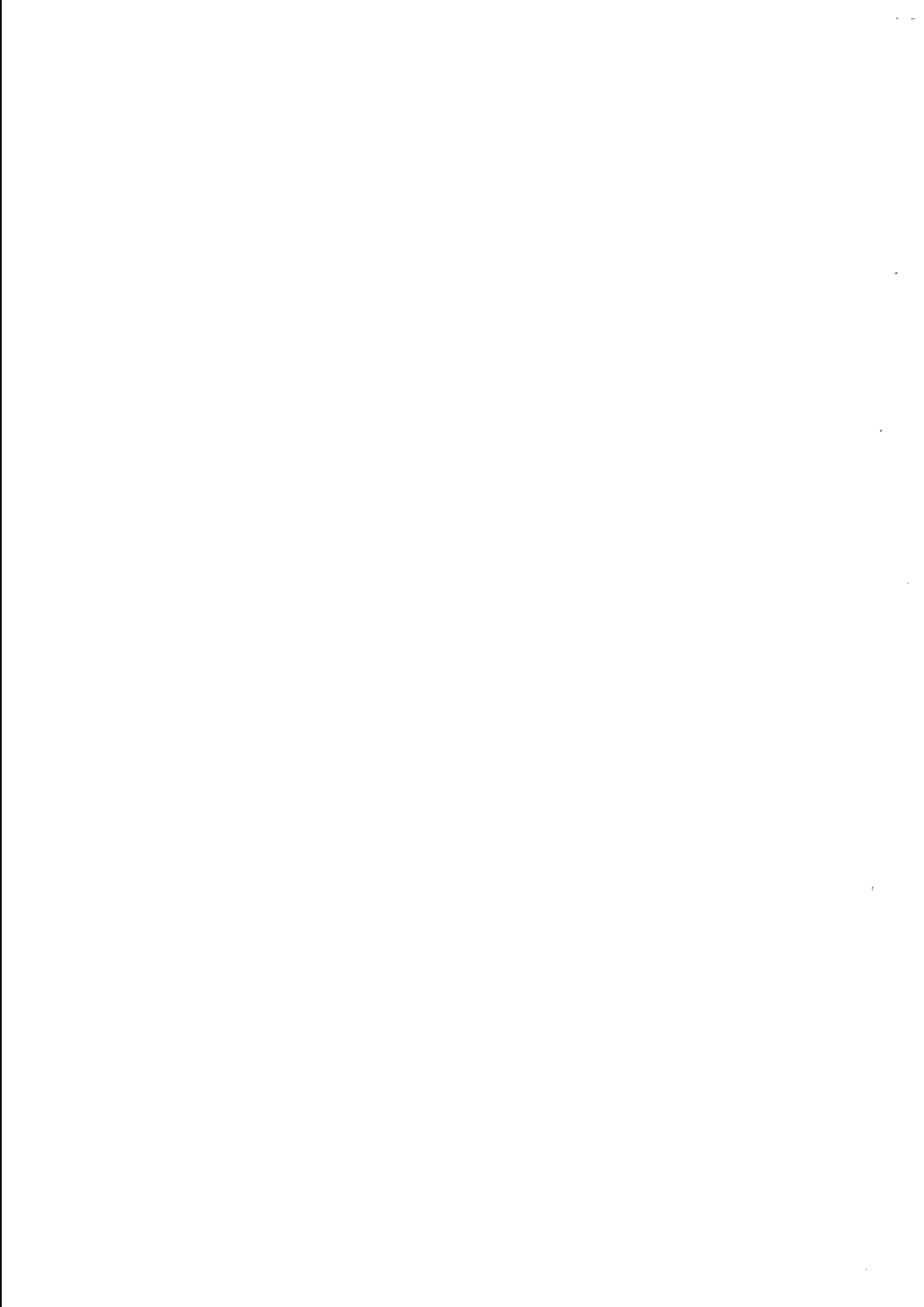
CODE	MEANING	DESCRIPTION
*R	Illegal relocation	<ul style="list-style-type: none"> – Either a predefined expression or predefined relocatable section has been input. – Too many relocatable symbols are added to or subtracted from each other. – The expression is equal to the result of a subtraction of a relocatable part from an absolute part. – If an external reference is specified the displacement value must be absolute. – The instruction code operation defined by an EQU directive must be absolute.
*S	Illegal statement	<ul style="list-style-type: none"> – 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 a 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. .
*X	Illegal expression	<ul style="list-style-type: none"> – 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 must not contain more than one term.
*****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.

2.3.4 Saving the Assembled Module

After a successful assembly the source module can be saved by using the command :

```
KPF L /S [ (file-name  
           module name) ]
```

If (file-name
module name) is not given, the module is catalogued with the name given in its IDENT statement.



3. TOSS SYSTEM START

3.1. General

System start is the initialisation process which prepares a PTS 6000 Terminal Computer for application program running. It comprises the following steps :

- Load the TOSS Monitor into memory.
- Load the application program into memory.
- Load the application program into memory.
- Set up the required Monitor tables (optional Monitor configuration)
- Set up the required CREDIT tables (if a CREDIT program is being used (see the CREDIT Programmer's Reference Manual, M04)).
- Activate the application task. If there is more than one task to be activated, this must be done by the application itself (only the 1st application task, or DEBUG, is started by the Monitor).

3.2. System Start Procedure

The Monitor, Monitor configuration data, and application can be loaded from cassette, disk, or flexible disk. Loading can be controlled from the full panel (if present) or from the SOP only. All the relevant procedures are described below.

3.2.1. From Cassette

It is possible to load the system from one, or two cassettes depending upon which option was taken during system Generation (see \$PCAS, DOS 6810 System Software PRM—M11). The first option, everything on one cassette, is :

- Monitor
- Application
- [● Monitor configuration data]

The second option, two cassettes, is :

- Monitor and application on the first cassette
- Monitor configuration data on the second cassette

The generation of the cassettes is described briefly in section 3.3 but a detailed description is given in DOS 6810 System Software PRM, M11.

The procedure for loading the system is :

1. Ensure that the power is switched on at the Terminal Computer and at each peripheral device.
2. Insert the Monitor cassette in a cassette drive.
- 3a. To load from the full panel, press the following buttons in sequence :
 - 1) Reset (RST)
 - 2) Master Clear (MC)
 - 3) Initial Program Load (IPL)
- 3b. To load from the SOP only, press the SOP IPL Switch.
4. Select the cassette drive containing the Monitor cassette by pressing SOP switch 1 (for the left-hand cassette drive) or SOP switch 2 (for right-hand cassette drive).
5. The Monitor and application program will now be read into memory.
6. If a second cassette containing the Monitor configuration data is to be loaded, mount this cassette in a cassette drive and press the appropriate SOP switch (1 or 2) when the first cassette has been read. The Monitor configuration data will then be read into memory and the required Monitor tables will be set up.
7. The application will now be executed unless DEBUG has been included. DEBUG has a higher priority than the application program (because it is part of Monitor) and it will stop as soon as it is loaded, waiting for the operator's command (see Chapter 4).

During loading, lamps 1, 2 or 3 may light up :

- Lamp 1 indicates that an application is being loaded
- Lamp 2 indicates an input error
- Lamp 3 indicates that the application is too large for the memory

3.2.2. From disk

The Monitor and application must be on the same disk, either cartridge or fixed disk. The procedure for loading the system is :

1. Ensure that the power is switched on at the Terminal Computer and at each peripheral device.
2. Insert the disk on a disk drive, press the START button and wait for the READY lamp to light up.
If loading is to be done from the fixed disk, a disk cartridge must still be loaded.
- 3a. To load from the full panel, press the following buttons in sequence :
 - 1) Reset (RST)
 - 2) Master Clear (MC)
 - 3) Initial Program Load (IPL)
- 3b. To load from the SOP only, press the SOP IPL switch.
4. Select the disk containing the Monitor by pressing SOP switch 3 (for the cartridge disk) or SOP switch 4 (for the fixed disk) Monitor will then be loaded.
5. If there is only one application program on the disk, lamp 1 will light up and the application will be loaded automatically.
6. If there is more than one application program on the disk, all the lamps will light up. Choose the application program to be loaded by pressing the appropriate SOP switch.
7. The application will now be executed unless DEBUG has been included. DEBUG has a higher priority than the application program (because it is part of Monitor) and it will stop as soon as it is loaded, waiting for the operator's command (see Chapter 4).

During loading, lamp 1, 2 or 3 may light up :

- Lamp 1 indicates that an application is being loaded
- Lamp 2 indicates that the application chosen by the SOP switch does not exist.
- Lamp 3 indicates that the application is too large for the memory.

3.2.3. From Flexible disk

The Monitor and application program must be present on the same flexible disk. The disk is produced by using \$PFLEX during System Generation (see \$PFLEX DOS 6810 System Software PRM M11). The procedure for loading the system is :

1. Ensure that the power is switched on at the Terminal Computer and at each peripheral device.
2. Put the diskette into a drive and shut the door.
- 3a. To load from the full panel, press the following buttons in sequence :
 - 1) Reset (RST)
 - 2) Master Clear (MC)
 - 3) Initial Program Load (IPL)
- 3b. To load from the SOP only, press the SOP IPL switch.
4. Select the flexible disk drive containing the Monitor and application by pressing SOP Switch 5, 6, 7 or 8.

switch 5 = flexible disk 1, multiplex channel
switch 6 = flexible disk 2, multiplex channel
switch 7 = flexible disk 1, programmed channel
switch 8 = flexible disk 2, programmed channel

Monitor will then be loaded.

5. If there is only one application on the disk, lamp 1 will light up and the application will be loaded automatically.
6. If there is more than one application on the disk, all the lamps will light up. Choose the application program to be loaded by pressing the appropriate SOP switch.
7. The application will now be executed unless DEBUG has been included. DEBUG has a higher priority than the application program (because it is part of Monitor) and it will stop as soon as it is loaded, waiting for the operator's command (see Chapter 4).

During loading, lamp 1, 2 or 3 may light up :

Lamp 1 indicates that an application is being loaded.
Lamp 2 indicates that the application chosen by the SOP switch does not exist.
Lamp 3 indicates that the application is too large for the memory.

3.3. Deferred binding of Monitor Configuration Data

3.3.1. General

The TOSS for a particular PTS 6000 Terminal System must be generated by the utility SYSGEN. This utility generates the TOSS Monitor on disk. The Monitor can be copied to cassette or flexible disk for the system start procedure. This copying is done by the catalogued procedure \$PCAS (for cassette) or \$PFLEX (for flexible disk).

If the programmer wants to test his program with Monitor configuration data supplied at run time (deferred binding) he can exclude the Monitor configuration data from the Monitor cassette at system generation time. The Monitor will be generated without configuration data but with the Monitor Configuration Program (MONCON) which will read the configuration data at run time. In this case the Monitor and application program must be on the same cassette. The instructions to generate a Monitor configuration data cassette are given below in paragraph 3.

MONCON will read the Monitor configuration data from the cassette and generate Monitor tables at the end of memory (higher addresses).

Only the drivers and tables which are included at system generation time may be configured by MONCON.

The use of the Linkage Editor, \$PCAS, and \$PFLEX is described in the DOS 6810 System Software PRM (M11).

3.3.2. Monitor Configuration Cassette

When the Monitor and application load module have been read into memory, control is passed to MONCON.

This program will read the Monitor configuration data and generate the necessary Monitor tables related to the hardware configuration.

To enable MONCON to set up the correct Monitor tables, the following data must be supplied as Monitor configuration data on the cassette :

- Terminal class identifier
- Number of work positions in the terminal class
- Priority level of the task
- Terminal device class
- Connection on the local/remote channel unit
- Special device classes

3.3.2.1. Generating the Monitor Configuration Cassette

The cassette containing Monitor configuration data is generated at the console typewriter with the aid of DOS 6810 control commands.

The procedure for generating a Monitor configuration data cassette is as follows.

First insert a cassette in one of the cassette drives, then key in the following sequence.

- (i) ASG□/E1, TY10
- (ii) REW□/03
- (iii) RDA□/0A
- (iv) Monitor configuration data
- (v) WEF□/03
- (vi) PCH□/0A
- (vii) WEF□/03
- (viii) REW□ 3
- (ix) ULD□ 3

Explanation :

- (i) Assign file code/E1 (source input) to the console typewriter.
- (ii) Rewind the left-hand cassette drive.
- (iii) Read data from the source input file (typewriter) and transfer to temporary disk file /A.
- (iv) The format of the Monitor configuration data is described below.
- (v) Write an end of file mark on the cassette.
- (vi) Write the contents of temporary disk file /A to the cassette, an end of file mark is written automatically.
- (vii) Write one end of file mark on the cassette.
- (viii) Rewind the cassette.
- (ix) Unload the cassette.

3.3.2.2. *Format of Monitor Configuration Data*

The Monitor configuration data must be keyed-in in the following sequence :

- Terminal class identifier (2 characters) followed by the number of work positions in the terminal class.
- Priority level
- Terminal device class followed by the connection on the local/remote channel unit.
- Special device class.
- EN End of task definition

a. The terminal class identifier and number of work positions have the following format :

terminal class identifier : decimal-integer CR LF

< terminal class identifier > is specified in the main module of the application program.

< decimal-integer > is the number of work positions in this terminal class.

b. Priority level has the following format :

decimal-integer CR LF

< decimal-integer > consists of two digits specifying the priority level of the task, which is normally 60.

A number lower than 60, means that the task has higher priority than the one with level 60.

c. Terminal device class and connection have the following format :

T decimal-digit, decimal-digit L CR LF
(local channel unit)

T decimal-digit, decimal-digit R CR LF
(remote channel unit)

< T decimal-digit > , is the terminal device class which is specified during system generation.

< decimal-digit L > or < decimal-digit R > is the number of the connector on the local or remote channel unit , respectively to which the work position is connected.

d. Special device class has the following format :

S decimal-digit CR LF

< S decimal-digit > is the special device class which is specified during system generation.

e. EN CR LF terminates the parameters for the terminal class concerned and the same parameters as mentioned above may be specified now for another terminal class.

When closing with ENEN, the data that follows will be considered as common device information. An example of Monitor configuration data is shown below for a configuration which consists of 3 work positions configured in the same way and one special work position with a different configuration.

Common devices are used by both terminal classes.

The first terminal class is T0 and the second is F0.

Monitor configuration data :

T0 : 3 (Three tasks with identifiers T0, T1 and T2 will be generated)

60

T1, 1L (Terminal device class 1 on channel unit line 1, 2, 3)

T2, 1L (Terminal device class 2 on channel unit line 1, 2, 3)

S1 (Special device class)

EN

F0

T3, 8L

EN

EN

S3 Common devices

S4

EN

3.3.2.3. *Errors During Configuration*

During Monitor configuration any errors will be indicated on the System Operator's Panel :

- Lamp 1 — Monitor configuration data reading
- Lamp 2 — Cassette input error
- Lamp 3 — Format error
- Lamp 4 — Memory overflow

4. ASSEMBLER DEBUGGING PROGRAM

During the production of a program, the programmer requires a method investigating errors that may have occurred. The Assembly Debugging Program (DEBUG) is provided to assist the programmer in monitoring and controlling the execution of this program in order to find errors.

This chapter contains :

- a description of the Assembly Debugging Program
- definitions of the control parameters
- instructions for running the program
- interpretations of the result.

4.1. Introduction

The Assembly Debugging Program (DEBUG) is an interactive diagnostic task which runs under the control of the TOSS Monitor. It runs in parallel with the Assembly application program being tested.

DEBUG may be used to control the execution of the application program in the following ways :

- the contents of registers or memory may be examined or modified;
- the application program may be stopped by inserting breakpoints at selected places, and started again;
- parts of the program may be conditionally executed, once or a number of times (looping);
- calculations on addresses may be performed.

DEBUG operates in either one of two modes :

- Debug mode – the programmer has control over the execution of the user program;
- User mode – the user program is executed normally.

Readers of Chapter 4 should be familiar with the following DOS 6810 System Software concepts :

- linkage editor
- control command
- user library
- TOSS system generation

These concepts are explained in the DOS 6810 System Software PRM (M11).

Before using DEBUG, the specialist reader should read Appendix A, Memory Organization.

4.2. Using DEBUG

DEBUG is specified at SYSGEN time and is part of the TOSS Monitor. If DEBUG is not required (e.g. for production versions of the application program) it must be excluded at SYSGEN time.

DEBUG runs on any level greater than or equal to 8 and always on a higher level than the application program being tested. The default value is 50 (/32). All Monitor tasks, except data management can use DEBUG. Control is handed to DEBUG immediately after the TOSS Monitor and application program are loaded into memory (system start).

The User communicates with DEBUG via the system operator's console (CTW) using the commands described in Section 4.

DEBUG maintains two sets of registers-relocation registers for addressing purposes and "pseudo registers" for processing purposes.

The use of relocation registers is described in Section 4.3.3.

The pseudo registers correspond to CPU registers P, A1 to A15 (inclusive).

These are loaded when DEBUG is entered from either a halt command or a trap.

The user may then examine and modify these contents, and before the user program is restarted, the pseudo registers are copied into the real CPU registers. From the user's point of view these are indistinguishable from the real registers.

4.2.1. Breakpoints

The user can stop his program at a specific point by a command to DEBUG. The point at which the program stops is called a 'breakpoint'.

DEBUG replaces the instruction at the breakpoint with an illegal code, causing a call to the trap call interpreter (/6000). When the program reaches the call, control is given to DEBUG. The user may then examine and modify registers and memory before continuing.

DEBUG also provides the possibility to loop through a breakpoint a specified number of times and to execute breakpoints conditionally depending on register or memory values. This can be useful when debugging program, loops etc.

It is possible to maintain up to sixteen breakpoints simultaneously. However, only one may be looped or executed conditionally at a time. This is called the 'active breakpoint'. The last breakpoint given or looped on is active.

When looping, DEBUG uses an internal register called the loop counter.

4.2.2. DEBUG Start

During execution, DEBUG is entered after a breakpoint, illegal instruction, or if the user issues a halt command from the console.

When DEBUG is started, relocation register E contains the first address of the application program.

DEBUG can be restarted from address /92.

4.3. DEBUG Input

4.3.1. General

DEBUG provides the programmer with a variety of commands to control the testing process. The commands are briefly :

Command	Mnemonic
calculate	=
open memory word	/
go	G
halt	H
interrupt	I
loop	L
print memory location	M
proceed from breakpoint	P
print location registers	Q
print CPU registers	R
open program status word	S
set breakpoint	T
verification	V
remove breakpoint	Y

These commands are described in full below :

4.3.2. DEBUG Modes

DEBUG operates in one of two modes known as B (debugging) mode, and U (user) mode. In B mode DEBUG is running and waiting for an operator's command. B mode is selected whenever the application program stops. The mode is selected by the system

A stop occurs when :

- a halt (H) command is keyed in
- a breakpoint is encountered in the application program
- a verification halt condition is encountered
- an illegal operation code is detected.

U mode is selected when one of the following commands is keyed in :

- proceed from breakpoint (P)
- loop through breakpoint (L)
- Go (G)

Commands other than H, P, L and G will not result in a change of mode. The current mode is indicated by the letter B or U printed at the left of each line of output. Immediately after system start DEBUG is in B mode.

4.3.3. Relocation Registers

DEBUG maintains 16 relocation registers. These registers are numbered 0 to F and may be used in commands as indexes when referring to memory locations. The contents of relocation registers may be examined and modified using the Q command.

The following description illustrates the way in which the relocation registers are normally used. The start address of the module currently being tested is loaded into a relocation register. Commands then refer to locations within this module by quoting the address relative to the start of the module (listed by the assembler as the second field from the left) together with the number of the relocation register.

Relocation registers may also be used to save constants.

When starting an application program, relocation register E contains the LOAD address of the program - 8. Register F contains - 8 which can be used to obtain module start addresses from the Linkage Editor Map.

4.3.4. Addressing

The following commands contain references to memory addresses :

G I L M P Q R S T Y = , / .

Relative, absolute or indirect addresses may be used in these commands. All address values are given as hexadecimal numbers.

The relative address is calculated as a displacement from a named relocation register. The address in the relocation register could be the start address of the module or some other reference address (the start of an array for example). The relative address from the start of the module is shown in the second field from the left on the Assembler listing. The relocation register may also contain an absolute address.

The absolute address is that which is given in the program counter and is the displacement from word zero of memory. If the address refers to the start of an array, a displacement value must be given to specify the word within the array (counting the first word as 1).

Indirect addresses are used to reference :

- CPU registers (R)
- control registers (I)
- relocation registers (Q)
- memory words (=, I, V)

If an address is indirect it must be prefixed by an asterisk (e.g. * 10). In this case the indirect address will point to a memory word that contains the absolute address to be referenced. If the resulting address is odd, the next lower even address will be used.

4.3.5. Command Syntax

Commands are keyed in immediately after the B or U prompt which is printed at the left of each line by DEBUG.

Commands have the following Syntax :

$$\left\{ \begin{array}{l} \text{B} \\ \text{U} \end{array} \right\} [\text{parameter 1}] [; \text{parameter 2}] \text{command}$$

"B" or "U" is printed by the system at the beginning of the line. "Command" is one of the single character commands listed above in section 4.3.1.

$$\text{parameter} ::= \left\{ \begin{array}{l} \text{TERM} \\ \text{parameter } (\pm) \text{ parameter} \end{array} \right\}$$

$$\text{TERM} ::= \left\{ \begin{array}{l} \text{hexadecimal integer} \\ * \text{ term} \\ \text{term R} \\ \text{term Q} \\ \bullet \text{ term} \end{array} \right\}$$

where : — "hexadecimal integer" is a four digit hexadecimal number

"* term" is an address of a memory word that contains the address of the required information.

"term R" is the sum of "term" and the contents of CPU register R.

"term Q" is the sum of "term" and the contents of relocation register Q.

"• term" is a register that contains the address of the required information.

An open and modify command has the following syntax :

$$\left\{ \begin{array}{l} \text{B} \\ \text{U} \end{array} \right\} /n\text{nnn} [\text{parameter}] \text{C}$$

where "/n\text{nnn}" is the content that "parameter" optionally replaces,

"C" is a terminating character which can be :

CR modify and close

LF modify and open the variable at the next higher address

@ modify the variable and use the new value as the address of the next variable to be opened (i.e. indirect addressing)

This next variable is automatically opened.

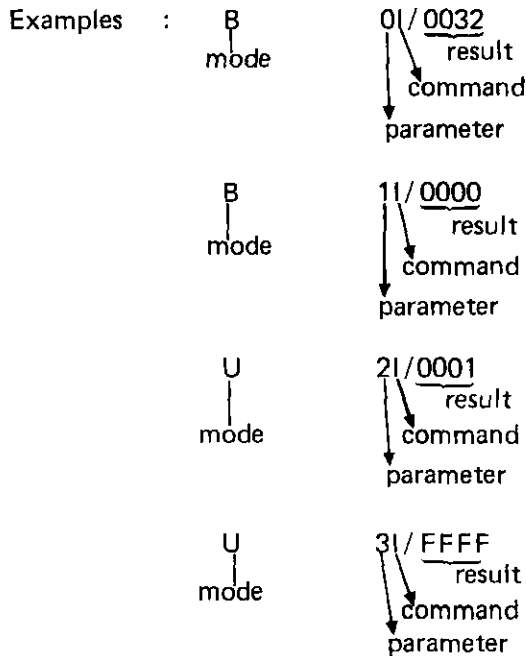
Any other character will result in the current variable being closed without modification and without the next variable being opened.

The description of each command will specify which terminating characters can be used.

Notes : An empty or non-existent parameter may have a special meaning to the command, see the description of each command.

Parameters are left-shifted so that only the last four character keyed in are taken as significant, the others are ignored. If an undefined command is keyed in, a question mark is printed and no further action is taken. If an illegal command is keyed in, it is rejected and 'NO' is typed out; no further action is taken.

Code	register
0	Priority level of DEBUG is displayed
1	DEBUG interrupt mode is displayed
	0 = inhibit mode (INM) non-zero = enable mode (ENB)
2	DEBUG interrupt control, 0 = None
3	Real time clock control, 0 = Off



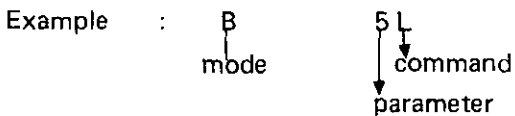
4.3.11. Loop (L)

Format 1 : parameter L

Description : Loops "parameter" times through the current breakpoint.
The program will run without stopping at that breakpoint until it comes to it (parameter + 1) times.

Format 2 : L

Description : Loops only once through the current breakpoint.



4.3.12. *Print Memory Location Command (M)*

Format 1 : parameter 1; parameter 2 M

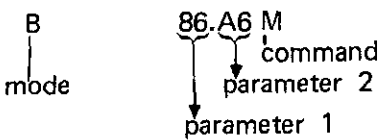
Description : Prints all memory locations from "parameter 1" to "parameter 2" (inclusive).

Format 2 : parameter M

Description : Prints eight words from "parameter".

Format 3 : M

Description : Prints eight words from the last address specified.

Examples :

The result is shown below

0086	0000	0236	23DC	FFFE	0000	5710	5714	10AC	...6#.....W.W...
0096	2276	01F7	0000	00F1	0000	0000	8120	0246	".....F
00A6	5710	85AD	0090	86AD	FFF8	9600	0094	8120	W.....

4.3.13. *Proceed from Breakpoint (P)*

Format 1 : parameter 1; parameter 2 P

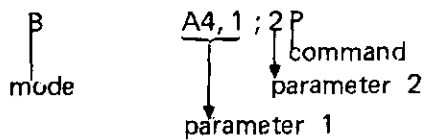
Description : Removes the current breakpoint, sets a new breakpoint "parameter 1", proceeds, and loops "parameter 2" times through the new breakpoint.

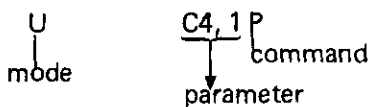
Format 2 : parameter P

Description : Sets a new breakpoint "parameter" and proceeds from there.

Format 3 : P

Description : Removes current breakpoint and proceeds.

Examples :




4.4. Running DEBUG

4.4.1. DEBUG Output

Each time DEBUG is entered, the following message is printed :

C — PC = LC, REL — PSW

where

C is used to indicate the way DEBUG was entered.
It may have the following values :

C	Meaning
S	start or restart
T	breakpoint
I	illegal instruction
H	halt command
PC	program counter is the absolute address where the program was interrupted
LC, REL	is the relative address LC is indexed by the relocation register REL. The REL with the nearest corresponding LC value is chosen.
PSW	CPU program status word at interrupt, see Appendix A

Output from the commands is shown in the paragraph describing each command in Section 4.3.

4.4.2. Loading Segments for Debugging

It is not possible to set a trap in a segment until it has been loaded. This means that it is necessary to set the trap in the load task just after the segment has been loaded, then find out in what partition the segment is placed and set a trap in the appropriate position. At the end of the load task an exit is made (LKM, DATA 3). The LKM is a useable breakpoint.

It is also possible to use the verify function (see Section 4.3.18) after setting a call in the load task. The segment pointer (address in the LSET to the segment) is found in the register A3 at exit of the load task.

4.4.3. Interrupt Control

When DEBUG is entered from a breakpoint, it is set to run at level 50 (= /32), in the mode ENB. If in INH mode, it is set to the same as the interrupted user program.

This is, in most cases, just what the user wants. INH sequences will not be interrupted, Monitor tasks will interrupt DEBUG etc.

However, there might be reasons to have DEBUG working in another way. For example:, when debugging synchronous devices, interrupt routines in ENB mode, time out functions, etc.

Therefore DEBUG has the following features :

for commands see Section 4.3.

1. The level may be changed. Since the console uses level 7, the level should not be set to less than 8.
2. The interrupt mode may be changed to either ENB or INH.
3. The function that *DEBUG* changes mode at breakpoints to the user mode may be excluded.
4. The real time clock may be handled by *DEBUG*.

4.4.3.1. *Interrupt Button*

Pressing the interrupt button (INT) on the full panel forces *DEBUG* immediately to command mode. This may be used to stop a long memory dump or to get the console on-line after pressing master clear (MC) —

The INT button is only available on computers with a full panel.

4.4.3.2. *Power Fail*

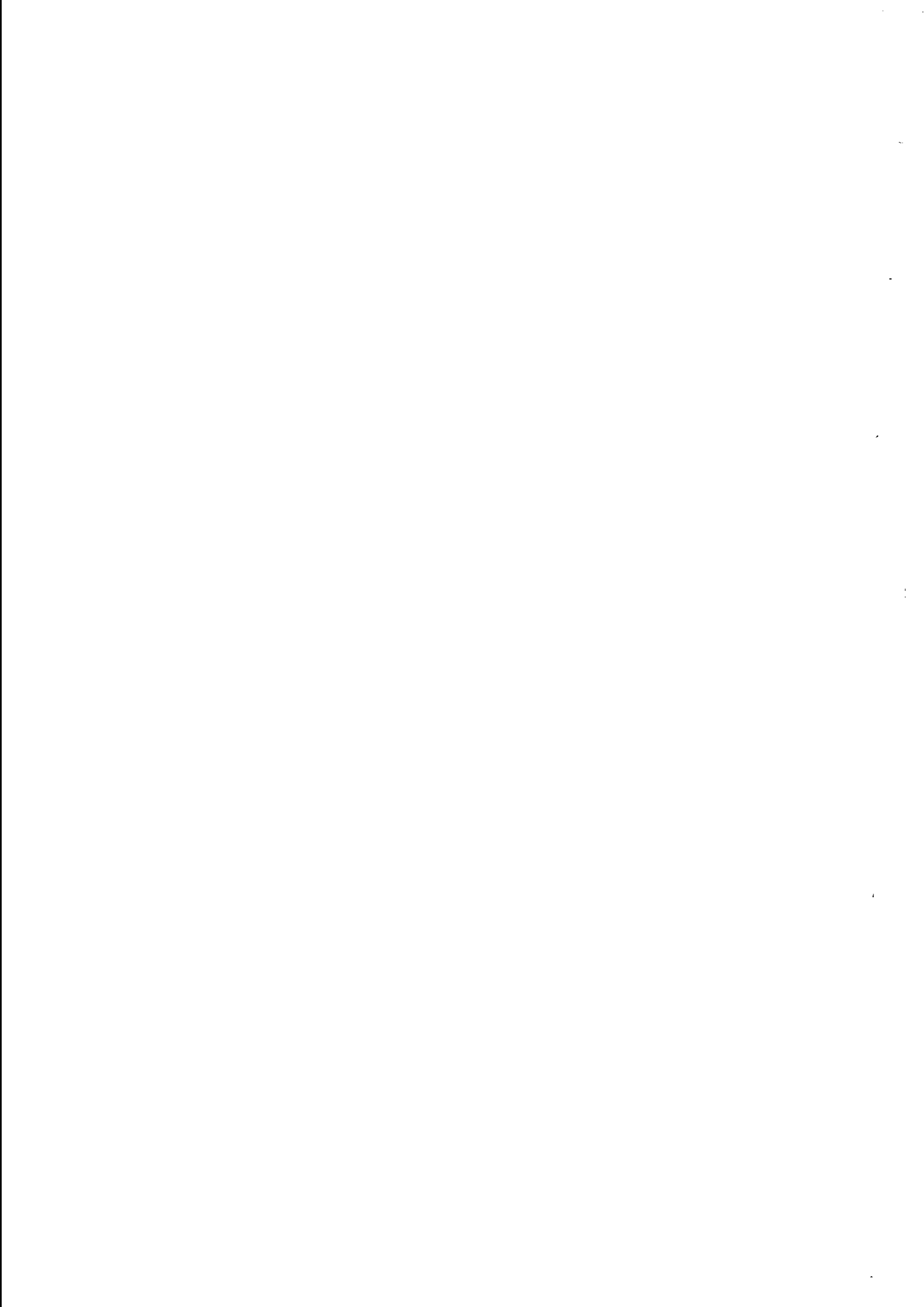
The power on/off interrupt is just switched through *DEBUG* after having set the console on-line.

4.4.3.3. *Real Time Clock*

The real time clock may be operated in two modes :

1. The RTC interrupt is switched through *DEBUG*;
2. *DEBUG* handles the RTC interrupt

Mode 1 is default.



APPENDIX A : MEMORY ORGANIZATION

The information in this Appendix will not normally be required by most Assemble Programmer's but has been included for specialist programmers.

A.1 Memory Layout

AREA	ADDRESS
Hardware Interrupt Locations	/0 /7C
Pointer to Subroutine Call Interpreter	/7E
System Halts	/80 /84
Communication Vector Table	/86 /98
Stack Overflow Area	/9A /100
Stack	variable
Monitor	variable
User Program Area	variable end of memory

Figure A1 Memory Layout

- Locations /0 to /7C are hardware interrupt locations. They are hard-wired to internal and external lines. Each location contains the address of the interrupt routine required to service the interrupt connected to that location. The interrupt connected to location /0 has the highest priority (level 0).
- Location /7E contains the address of the subroutine call interpreter which handles simulation of certain instructions not included in the hardware.
- Locations /80 to /84 contain the system halts.
- Locations /86–/98 are the Communication Vector Table. This is a System table. This table has the following layout :

	AREA	CONTENTS
CVT without Memory Management	86 { CVT	Memory size
	CVT STB	A15 stack base
	CVT SBA	Start of buffer area
	CVT EBA	End of buffer area
	CVT INP	Interpreter table address
	90 { RF INIT	Jump to restart module
Memory Management	92 { RF BUGG	Jump to Debugger
	CVT APA	Application address
	CVT APS	Application Start address
	98 { CVT CLK	Real time clock
	CVT LSB	Address to LSBT
	CVT DK	File code start up disk
	FREPAR	Free partition pointer
	PARLEN	Length of partitions (in bytes)

Figure A2 Communication Vector Table

Notes

1. If Memory Management is used, the CVT extends into the stack overflow area.
2. CVT SBA holds the address after the root.
3. CVT APA is the address of the beginning of the root.

— The area occupied by the stack is defined at system generation time. When an interrupt occurs, P-register, PSW and a number of registers are stored here. The number of registers stored depends on whether the interrupt routine servicing the interrupt runs in inhibit mode (anywhere from 0 to 15 registers) or in enable mode and branches to the dispatcher (always 8 registers).

The A15 register always points to the next free location in the stack (where all information is stored towards the lower memory address).

When A15 reaches the value /100 or becomes lower a stack overflow interrupt is given.

— The area after the Monitor area is the user area.

A.2. 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 at SYSGEN time.

interrupt level	hardware interrupt location
0 to 63	/0000 to /007C

where level 0 has the highest priority and 63 the lowest. The levels are defined at SYSGEN time.

PROGRAM STATUS WORD

The program status word (PSW) contains data relating to the status of that program. It is maintained by the System Software.

These fields can be modified by the user	{	0 — 5	priority level
		6 — 7	condition register
These fields cannot be modified by the user except during Debugging	{	8	run indicator
		9	interrupt enabled indicator
		10	control panel interrupt
		11	power failure
		12	real-time clock (RTC)
		14 } 15 }	not used

