MicroByte

Software for the 2650

ASSEMBLER

EDITOR ASSEMBLER FOR THE 2650

by Ian Binnie and Martin Hood

Copyright MicroByte, October 1979.

CONTENTS

1. GENERAL DESCRIPTION

1.1 EDITOR

This is fundamentally a 'line oriented' editor with many features to facilitate the creation and editing of text files. Character oriented editing within a line is also supported. Files may be saved to, or loaded from cassette with the save and load commands. Multiple files may be present in memory at one time and be opened for use by specifying the start address of the file. Lines may be copied and appended to a secondary file, or the contents of secondary file merged into the primary file. A specific line in a text file may be accessed by relative movements forwards or backwards, by going to a specific line number or memory address, or by searching for a specific string of characters. Lines may be inserted, deleted, appended to or have strings within them replaced. Editor commands may be typed in either upper or lower case.

1.2 ASSEMBLER

The assembler is a two pass assembler which uses the standard SIGNETICS 2650 mnemonics. It is co-resident in memory with the editor and may be called from it for immediate assembly of a file. The assembler will support up to 1024 six character symbolic names and multiple pass assemblies. Object code may be directed to memory or to tape, and source listings to either the terminal or to a printer.

1.3 SYSTEM REQUIREMENTS

Hardware 2650 processor

Memory
Read/write memory 450 - 5FF for scratchpad.
Read/write memory or EPROM containing the program between 600-1BFF.
Read/write memory 1C00 - 1FFF for symbol table. *
Read/write memory 2000 - 2FFF for text buffer. *

Terminal - preferably a memory mapped terminal.
eg. DG640

Optional Printer - user must provide driver software.

Cassette system using standard BINBUG, PIPBUG or ACOS format.

Software
CHIN Keyboard input subroutine.
COUT Terminal display output subroutine.
POUT Printer output subroutine.
PINT Printer initialise subroutine.

Tape routines to suit system. See 'personality module' for details. Routines are provided for BINBUG, PIPBUG and ACOS formats.

* Note:- These storage allocations are the minimum recommended. Both the symbol table storage and text buffer may be positioned anywhere in memory. As supplied the symbol table will store up to 128 symbols. This is sufficient for programs which will fit within the small text buffer allocated. Refer to appendix e. 'Adapting to Your System' for details about reassigning the symbol table and text buffer storage if your system has more memory available.

1.4 PROGRAM START

The cold start for this program is located at 600. Typing G600 will commence execution of the program.

The warm start entry form this program is located at 603. This entry bypasses the editor initialisation process.

Section 1. - TEXT EDITOR

2. THE TEXT FILE

2.1 FORMAT OF A TEXT FILE

Each text file created by this editor begins with an STX marker, <02>, and ends with an ETX marker, <03>. The STX remains fixed at the address specified when the file is first created, this defaults to 2000, but the ETX marker will move as the file varies in length as lines are added or deleted. Each line of text is delimited by a carriage return (0D). As well as any printing character, linefeeds (0A) and TABS (09), may be entered in the text. TAB characters are expanded to spaces by the output driver routine used by the editor.

FORMAT <02> <0D> <text> <0D> ....<text> <0D> <03>

2.2 THE CURRENT LINE

This editor is oriented towards editing text files a line at a time. It has a pointer which points to the start of what is called the 'CURRENT LINE'. The current line pointer (CLP) may be moved around and be made to point to the start of any line in the text file. The current line is particularly important because all editing commands act either on the current line, or relative to its position. The current line pointer indicates the start of the line which has just been displayed on the terminal.

2.3 CONTROL CHARACTERS

The editor will respond to the following control characters.

HTAB (CTL I) Advance to the next horizontal TAB position.

BS (CTL H) Backspace and delete previous character.

CTL R Repeat last editor command.

DELETE Delete whole line.

The horizontal tab character (CTL I), should be used to format assembler source code. The editor will expand tab characters into spaces so that tabbed text will line up in columns. The assembler will accept either single or multiple spaces, or a TAB character as the delimiter between fields, but single spaces will not be expanded to produce a source listing which is formatted in columns.

Linefeed characters may be inserted into the text file, but any other control characters will be ignored since it is very

likely that non-printing control characters could be inserted which would lead to errors which would be very difficult to find.

The BACKSPACE key (Control H) may be used to move backwards along a line to make corrections, or if the line is a total mess, it may be deleted with the DELETE key.

CONTROL R, the control key and R pressed at the same time may be used to repeat a previously executed command.

2.4 INTERUPTING A LISTING

Pressing the BREAK key will interupt any listing in progess and abort to the text editor command level. The listing of a long text file using the LIST command may be terminated with the BREAK key. Similarly an assembler listing either to the terminal or to a printer will be aborted by pressing the BREAK key.

A BREAK is tested by checking the status of the sense bit whenever a character is output to either the terminal or the printer. Thus any listing may be interupted, but the output of object code to memory or tape cannot be interupted unless listing is occurring as well.

Refer to appendix e, if your system does not input characters through the sense input.

3.0 EDITOR COMMANDS

3.1 CREATING A FILE

Upon entry to the editor, the first instruction to be executed must be one of the following to 'open a file'.

NE or NE.<address>	opens a NEW file at the specified address.
O or O.<address>	makes an OLD file, currently in memory at the address specified, available for access by the editor or assembler.
LO	LOADS a file from tape into memory beginning at the start address of the currently open file and then opens it for use.

If no address is specified the file will be opened at the default address specified in the personality module. Currently this is 2000.

3.2 INSERTING LINES OF TEXT

The command 'I' will place the editor in the mode to INSERT lines of text into the file. This mode is indicated by the presence of a '-' as the prompt instead of the normal '>'. Once in the insert mode, lines of text may be typed in, terminated by a return at the end of each line. Any line inserted will always be inserted BEFORE the current line. The INSERT mode may be exited by typing two carriage returns in succession, in other words, a null line.

If a line is entered which exceeds the length of the input buffer, 127 characters, an error message *O* will be displayed and the line will be ignored.

If a line is inserted which would cause the text file to exceed the allocated memory buffer size, an error message *M* will be displayed and the line will not be inserted.

3.3 MOVING AROUND THE TEXT FILE

Once all lines of text have been entered into the file using the INSERT mode, the CURRENT LINE POINTER points to the end of the file. The current line pointer always points to the line which has just been displayed on the screen and each of the followins commands will display the line reached. These five commands allow quick and easy movement around the file.

T	moves the CLP to the TOP of the file.
N,n	moves the CLP past the NEXT n lines.
B,n	moves the CLP BACK n lines.
G,n	GOES to the nth line in the file. This command is useful for quickly accessing lines which the assembler has diagnosed as containing an error.
G,<address>,n	GOES to the address specified and lists n lines. This command is useful for checking secondary files.

A carriage return will advance to the next line and list it. The easiest way to move short distances forward through the file while listing lines, is to repeatedly press the return key.

3.4 LISTING LINES

L	LISTS the current line.
L,n	LISTS n lines of the text file starting at the current line. The current line pointer is not moved by this command. After the listing is complete the current line pointer will still point to the first line listed.
L*	LISTS ALL lines. The qualifier '*' may be used in many commands in place of ',n'. It is interpreted as meaning "ALL occurrences" from the current line to the end of the file.

3.5 DELETING LINES

Once a line containing an error has been located by moving around the file, often the simplest way to correct the error is to DELETE the current line and INSERT a new one in its place.

D	DELETES the current line.
D,n	DELETES n lines starting at the current line.
D*	DELETES ALL LINES from the current line to the end of the file. Use this command with care.

3.6 APPEND

A/string/	APPENDS the string between the delimiters to the end of the current line. The delimiter does not necessarily have to be a slash. Any character which is not contained in the string will suffice. This command is useful for appending comments to the end of an uncommented line of assembler source code.

3.7 CHANGE

C/string1/string2/	Searches the current line for the first occurrence of 'string1' and CHANGES it to 'string2'. The strings do not have to be the same length. The delimiters, here '/', can be any character not contained within either string.
C/string1/string2/,n	CHANGES the FIRST occurrence of string1 in each of the next n lines.
C/string1/string2/,n*	CHANGES ALL occurrences of string1 in each of the next n lines.

'n' must be less than 256 for this command, but the command may be repeated by typing Control R.

3.8 FIND STRINGS

A particular string of characters in a file can be found using the 'F' command.

F/string/	FINDS the first occurrence of the specified string of characters, starting with the line following the current line. The line containing the string is listed if it is found. Further occurrences of the string may be found without retyping the command by using Control R to repeat the original command.

3.9 THE 'X' COMMAND

The 'X' command is a particularly powerful feature of this editor as it allows editing on a character-by-character basis within a line of text. Once the 'X' command has been invoked, a number of control characters become effective for editing single characters within the current line.

Linefeed	moves the cursor FORWARD one space.
Backspace	moves the cursor BACKWARDS one space.
Delete	DELETES the character currently under the cursor and closes up the line on the screen.
Control X	DELETES ALL characters from the current cursor position to the end of the line.

A Tab character, or any printing character which is typed in, will be inserted into the line of text BEFORE the current cursor position. TAB characters embedded in the line will be displayed as an underscore,'_'.

The advantage of this command is that it allows the editing of characters one at a time and immediately displays the results of the changes on the terminal. This command will work most effectively with a memory mapped display such as a DG640, and is not recommended for terminals which communicate with the computer at a speed less than 120 characters per second since the long time taken for the line to be updated may prove annoying.

4.0 SECONDARY FILE COMMANDS

The principal use of a secondary file is to enable a number of lines to be moved from one place in a text file to another. In this case the secondary file is being used as a temporary storage area for the lines being moved.

Alternatively, the secondary file may be used to collect together lines from various parts of the primary file, since the COPY command copies by appending lines to the end of the secondary file. The following example illustrates these applications.

PROBLEM: To move twenty lines of text from one place to another in a text file.

C0,20,3C00 COPIES twenty lines of text, starting at the current line, into a secondary file at address 3C00. A NEW file will be opened if no file exists at 3C00. However, if a secondary file exists at this location, the lines will be added after those already present in the file. The address of the secondary file should be chosen so that it does not overlap the end of the primary file, or another secondary file. The limits of the currently open file are displayed on the terminal by the S command. The current line pointer is not moved by this command.

D,20 DELETES from the primary file the twenty lines just copied into the secondary file.

Position the Current Line Pointer to display the line before which you wish to insert the twenty lines just copied into the secondary file.

M,3C00 MERGES the secondary file at 3C00 into the primary file BEFORE the current line and inserts these twenty lines back into the primary file in their new position. The current line pointer remains pointing at the line after the inserted lines at the completion of the merge.

MERGE may be used to insert many copies of a secondary file into the primary file, since the secondary file is not affected by the MERGE operation.

Caution - Space is first made in the primary file and then the secondary file is inserted into the space made. A sufficient gap must exist between the primary and secondary files so that the primary file does not overwrite the secondary file during this procedure. The memory area occupied by the symbol table during the assembly process may be used for secondary file operations.

5.0 LOADING AND SAVING FILES

Files may be SAVED onto tape using the SA command or LOADED from tape using the LO command. The precise format of the files on tape and the commands used will depend on which tape I/O routines are installed in the personality module associated with the program.

5.1 BINBUG and PIPBUG VERSION

Files are saved on tape as a memory image commencing with <02> and concluding with <03>. The precise format is specified in the personality module documentation.

The loader routine will load a file in the dumped format at the address of the currently open file.

5.2 ACOS VERSION

Files are saved on tape as assembler source type files including the STX and ETX markers. Standard ACOS format is used. The normal ACOS command suffixes apply, T, L, N and F, but the prefixes are different, SA for 'SAVE' and LO for 'LOAD', instead of D and L respectively. For example:

LOT FRED	LOADS the text file 'FRED' from tape into memory starting at the address of the currently open file.
SAT EDITOR	SAVES the text file currently open in memory as a text type file with the filename 'EDITOR'.
SAN	TURNS ON the DUMP cassette recorder.
LOF	TURNS OFF the LOAD cassette recorder.

6.0 EDITOR UTILITY COMMANDS

STATUS

S — Displays the STATUS of the currently open file to the terminal in the following format:

2000	Address of start of text marker <02>.
2142	Address of the start of the current line.
3F76	Address of the end of the file <03>.

EXIT

E — EXIT from the editor via the Jump at 606. Currently this vectors to 22.

E.<address> — EXITS from the editor to the specified address.

CONTROL R — The control key and R depressed together results in the last command executed being repeated. The main use for this function is to FIND later occurrences of strings with the F command without having to retype the whole line. It will also function with the CHANGE command to change more than 255 lines at a time. Control R must be the first and only character typed in on the line for it to repeat the previous command. Control R will function with any command except X, SA and LO because these commands destroy the contents of the input buffer.

AS — ASSEMBLE the source file currently open by the editor. See Section 2. for details about the assembler.

Section 2. - ASSEMBLER

7. ASSEMBLER DESCRIPTION

This assembler can be made to assemble the file currently 'open' in the text editor with the AS command.

7.1 USING AN ASSEMBLER

It is assumed that the reader is familiar with the 2650 instruction set and the standard MNEMONICS assigned by SIGNETICS to describe each instruction. These are described in the 'Signetics 2650 Programming Manual'.

If you have written any programs in 'machine code', then it is most likely that you have already performed manually most of the functions that an assembler does automatically. The activity of writing those programs was a form of 'assembly by hand'. If you were to write a routine to input a character from the keyboard and echo it to the terminal, it is most unlikely that you would immediately write out the following machine code.

```
3F 02 86
3F 02 B4
1B 78
```

If you were writing this code, you would have thought in terms of a set of instructions something like this:

```
Branch to a subroutine to input a character
Branch to a subroutine to output the character
Branch back to input another character
```

Or maybe you thought out the program in standard mnemonic form like this:

```
BEGIN   BSTA,UN 286
        BSTA,UN 2B4
        BCTR,UN BEGIN
```

If this is the way that you 'hand assemble' code, then you already know the fundamentals of writing assembler source code, because this is precisely what an assembler does. It translates these mnemonic codes into 2650 machine code. However an assembler also does much more than this.

7.2 WHAT DOES AN ASSEMBLER DO?

1. An assembler translates English like mnemonics into machine instructions. For example:

   eg. BSTA,UN becomes 3F
   LODI,R0 becomes 04

2. An assembler allows symbolic labels to be used in place of numbers. A symbolic label is simply a name which corresponds to a numeric value.

   There are two advantages to be gained by using a symbolic label instead of its corresponding numeric value.

   The number referred to by the label is assigned the more easily understood name of the label. Using the label TAB indicates the purpose of the quantity more clearly than its equivalent numeric value of 9.

   A label is not fixed in value but is evaluated and has this value assigned to it each time the assembler is run. To change the value of a label, it is only necessary to change its value in the one place in the source program where it is defined. The assembler will use the current value of the label wherever else in the program that label is used.

3. The assembler calculates relative addresses each time the source code is processed and automatically adjusts these branches.

4. The assembler does simple arithmetic and base conversions.

5. A 'plain language' listing is produced which is much easier to understand and interpret than a list of object code.

7.3 ASSEMBLER PASSES

This is a 'two pass' assembler. This means that it scans the assembler source code twice to produce its output.

PASS 1

During the first pass, the assembler scans the source code and builds up a table of symbols and their values.

Each symbol must be defined once and only once on the left hand side of an instruction or a pseudo-instruction. The value assigned to the symbol will usually be the address in memory of the first byte of the instruction which the label precedes, with one exception. The statement:

```
LABEL  EQU    20
```

assigns the value of 20 to the symbol LABEL. The EQUATE pseudo instruction makes the value of a symbol equal to the value of the expression on the right side of the instruction.

If an attempt is made to define a symbol more than once, then its value will be ambiguous. The assembler cannot know which of the values it should use and will produce a DUPLICATE SYMBOL ERROR. Some SYNTAX ERRORS will also be detected on the first pass.

PASS 2

During the second pass the source code is scanned again from the beginning. This time the assembler has a list of values in its symbol table to substitute for a symbolic label wherever one of these appears on the right side of an instruction. It is during the second pass that the object code is produced. Object code cannot be produced during the first pass because the values of many of the symbolic labels are not known when they are required because they are forward referencing. Source listings are also produced during the second pass.

ERRORS

Most of the errors which can be made in assembler source code are detected during the second pass.

If a symbolic label is never defined because it does not appear on the left side of an instruction, then the assembler cannot know what value to assign to the label when it appears on the right side of an instruction. This will cause an UNDEFINED SYMBOL ERROR to be generated. A common cause of undefined symbols is spelling the label name incorrectly.

If a relative branch is made which branches beyond the -64 to +63 byte limit, then a BRANCHING ERROR will be generated.

If an instruction is not a valid 2650 instruction or assembler pseudo-instruction, then a SYNTAX ERROR will be generated.

Most errors become obvious upon inspection of the code. However, the fact that a program assembles without errors does not guarantee that it is logically correct. Error free assembly only guarantees that the program is syntactically correct and does not violate any 2650 language rules, and that the assembler is able to produce machine code unambiguously.

OUTPUT

The assembler may output object code to memory or to tape or it may not output object code at all. The assembler may also output a listing to the terminal or to a printer. This listing consists of the object code as well as the source code which produced it.

8.1 FORMAT OF A LINE OF ASSEMBLER CODE

Each line of assembler 'source code' consists of one or more sections or 'fields'. Each field is separated from the next by a 'delimiter' which can be either a space or TAB character. The use of a TAB character (Control I) is recommended since only a TAB will be expanded to form columns on the source listing output. The fields in a line of source code are as follows:

[label] <instruction <operand> [comment]

for example:

```
LOOP        BSTA,UN       CHIN        INPUT CHARACTER
```

8.2 LABEL FIELD

This is a symbolic pointer used to identify the address of a particular point in the program. This may be:

- the destination of a branch.
- a reserved storage location.
- a byte of data.

A label:

- must be composed of either letters or numbers.
- must start with a letter.
- must be unique in its first six characters.

The folowing are examples of valid labels.

```
FRED    CHIN1    DEL25    A
```

8.3 INSTRUCTION FIELD

The instruction field contains a mnemonic which is used to specify a machine language command. A mnemonic is used because it is easier to remember an instruction if it is called by its mnemonic name rather than its numeric representation. For example:

```
LODI
STRA
BCTR
```

Most instructions also include a resister or condition code specification, which follows the instruction, but is separated from the instruction by a comma. For example:

```
ADDI,R0
BCTR,EQ
```

The instruction field may also contain a directive to the assembler to perform some function. These instructions are normally called 'PSEUDO INSTRUCTIONS', since they are not translated into 2650 machine code. For example:

```
PAG
ORG
RES
```

A full list of pseudo instructions is included in Appendix d.

8.4 OPERAND FIELD

Most instructions require an operand field. The value of the second and third byte of an instruction which results in two or three bytes of object code is determined by the the value of the operand field in the source code. An operand consists of a symbol or a constant, or an expression which is a combination of symbols or constants by addition or subtraction. For example:

```
TEMP
BUFF+1
A'Z'
H'FF'
TEMQ+LEN-1
19761
```

Branching and memory reference instructions may be preceded by an asterisk to indicate that the reference is to occur indirectly through the two bytes at the operand address. For example:

```
*TEMP
```

The operand of absolute addressing memory reference instructions may be followed by an index resister specification and this may be followed by a '+' or '-' to indicate auto-increment or auto-decrement. There is no comma between the resister and the + or -. For example:

```
SPC,R2+
BUFFER,R1
```

The operand field may not contain emebdded spaces.

8.5 COMMENT FIELD

This is an optional field which is not used by the assembler in the assembly process. Comments may be written in this field to help document the program. A line which commences with an asterisk will also be treated by the assembler as a comment line. For example:

```
*THIS IS A COMMENT LINE
```

8.6 NUMERIC CONSTANTS AND DATA TYPES

A constant may be translated into binary by the assembler from one of three data types. These are:

Decimal numbers
Hexadecimal Numbers
ASCII character strings

DECIMAL

If no data type is specified, the constant is assumed to be a decimal number. This default condition may be altered. For details see appendix f. Alternatively the prefix 'D' may be used. The following are all valid decimal constants.

```
9       99      D'56'   -1
```

HEXADECIMAL

Hexadecimal constants are specified by the prefix 'H'. The following are valid Hexadecimal constants.

```
H'FF'  H'9B'   H'D'    H'D,A,A,O'
```

ASCII STRINGS

ASCII character strings may consist of any printing ASCII characters. The following are valid ASCII strins constants.

```
A'PLEASE ENTER A NUMBER BETWEEN 1 AND 9'
A'^'
A'''
```

An ASCII strins containing more than one character is a shorthand notation allowed only in DATA statements.

9.0 ASSEMBLER OUTPUT OPTIONS

The utility of this assembler is enhanced by the provision of a number of output options.

9.1 NULL OPTION

If a carriage return is entered in response to the first 'OPTION?' request, then the source file will be processed by the assembler, both pass 1 and pass 2, and any lines which result in assembly errors will be listed to the terminal. The error line output to the terminal will specify an error code, the line number and the source code for that line. Error lines will always be listed to the terminal irrespective of which output option selected, however there is not much point in generating any assembly output until the source file is free of syntax errors since the object code produced will be incorrect.

Errors are listed as they are detected by the assembler. This results in all lines containing uniquely first pass errors such as duplicate labels (D), label errors (L), and symbol table full (F), being listed before the lines containing second pass errors. Some lines may generate errors on both passes for different reasons.

The format of the error line is:-

<error code> <source line number> <source code>

for example:-

```
U  0026        :        LODA,R0 TEMP       RESTORE REGISTER 0
```

This error line shows that TEMP is an undefined symbol at line 26. A table of error codes is provided in Appendix C.

The line in error may be quickly accessed using the G,<line number> command. The number of errors detected is also listed at the end of the assembly. Hopefully, but rarely, this will be zero.

With the null option, no output is generated. Only errors are listed. Once the souce code is free from assembly errors, any one, or a combination of the following options may be used to generate assembler output.

9.2 MEMORY OPTION M.<address>

The memory option, M.<address>, will cause the assembler to place object code into memory starting at the specified address. If no address is specified or if the address is zero, the object code will be stored in memory as directed by the origin statements in the source code. The main use of this ability to displace the object code, is to place the assembler output into a buffer for burning EPROM's. The displacement address does not alter any of the object code, so the displaced program will not work in its displaced location if it contains any absolute addresses. The displacement of output is ALL relative to the address of the first byte of object code output by the assembler. The difference between the actual address and the address specified in the option statement is calculated for the first byte output, and this displacement is added to all subsequent bytes. If your program contains more than one ORG statement, the object code following the second ORIGIN will be stored with the same relative displacement as the first section.

The assembler will not stop you from assembling output which destroys the assembler, so be careful. Assembling into memory a program which produces output between 440 and 1FFF, may crash the assembler by overwriting it. Overwriting the source file will also crash the assembly process.

MORAL:- Use the memory option with CARE.

A preferable method is to use the TAPE output option to generate an object tape and then reload this into memory.

9.3 TAPE OUTPUT OPTION T

This option will cause an object tape to be generated. The precise format of the object tape will depend upon the personality module in use with the program. For BINBUG systems the tape produced will be identical in format to that given by the DUMP command, binary format at 300 baud via the flag terminal. However the object tape produced by the assembler will have rather lengthy inter-block gaps. These are produced because the assembler must do all the second pass processing of a block of object code before it can be output to tape and this may result in a gap of about half a second between blocks. Listing and printing options should not be selected with the tape option unless the recorder's motor is controlled by the computer since this will result in very long times between object blocks being output.

An 'End of Tape Block' will be output when an END statement is processed, so only the final segment of a multiple pass assembly should conclude with an END statement. An operand following the END statement will be evaluated and recorded on the object tape as an execute address.

9.4 LIST AND PRINT OPTIONS L,n and P,n

The list and print options provide a means of obtaining a formatted assembly listing, either to the screen for a quick inspection, or to a printer for hard copy. Both options may be used simultaneously if it is required to display a listing as it proceeds on a remote printer.

Listings are formatted, with a heading, into numbered pages. The maximum length of the heading is 48 characters. Longer titles will be truncated. The number of lines of assembly code on each page is currently 55 lines which gives an overall page length of 66 lines including header and footer. The page length may be changed. See Appendix e. 'Adapting to your system' for details.

The listing or may be commenced part way through by entering P,n or L,n as the option, where n is the number of the first page to be listed.

9.5 ZERO OPTION Z

This option allows a mistake made while entering output options to be corrected. It ZEROS any output options already selected and allows you to choose again. The Z command does not reset the assembler pass options 1, 2, I or B.

9.6 MULTIPASS ASSEMBLY OPTIONS. 1, 2, C and B.

Four options are available to implement multipass assemblies. These are 1, 2, C and B.

OPTION 1 — If option 1 is selected, the assembler will only execute its first pass in which the symbol table is built up. No output will be generated except for error messages. This option should be used only for the first segment of the first pass of an assembly. Any other segments should be loaded into memory, the file opened and the C option used to CONTINUE with pass 1. Option 1 will result in the symbol table being cleared, while the C option will not do this but add symbols to the existing table.

OPTION 2 — Once all segments of the first pass have been processed, rewind the tape containing the source code, reload the first segment and open the file. However this time, assemble using option 2 which causes the assembler to execute only the second pass of the assembly. At this time, whatever output options are required should be selected. Again, subsequent segments should be assembled using the C option. Only the last segment should conclude with an END statement to ensure that listings and object tapes remain continuous.

OPTION C — Once a complete first pass has been run to build the symbol table, as many second passes as are required may be run without it being necessary to perform another first pass. Make sure that the first segment of the second pass has option 2 selected and that further segments use the C option to Continue. Otherwise the assembler will execute a first pass on the segment and destroy the orisinal symbol table.

OPTION B — May be used in conJunction with either option 1 or 2. Subsequent blocks of source code will be loaded from tape under program control until an END statement is encountered, or an assembly error is detected. This option can only be used if a tape system which may be started and stopped under program control is being used. The B option must be entered AFTER option 1 or 2.

9.7 PREDEFINED SYMBOLS OPTION I

The assembler contains a number of predefined symbols which are automatically written into the symbol table at the beginning of the first pass of an assembly. These are :-

Registers	R0	0		
	R1	1		
	R2	2		
	R3	3		
Conditions	Z	0	Zero	
	P	1	Plus	
	N	2	Negative	
	EQ	0	Equal	
	GT	1	Greater than	
	LT	2	Less Than	
	UN	3	Unconditional	
	NE	2	Not Equal	(TMI only)
PSW	CAR	1	Carry	
	LCOM	2	Logical Compare	
	OVF	4	Overflow	
	WC	8	With Carry	
	RS	10	Register Select	
	FLAG	40	Flag	
	SENS	80	Sense	

These predefined symbols may be disabled with the I option. This MUST be selected AFTER pass 1 has been selected on the first pass. Remember, however that if you disable the symbols, you will need to define separately some of the more common definitions such as registers and condition codes or you will end up with a large number of 'undefined symbol' errors.

9.8 MAXIMUM SIZE OF ASSEMBLIES

The fundamental limit to the size of a program which can be assembled by the assembler is determined by the number of symbols used. As supplied, the assembler has sufficient memory allocated to the symbol table to assemble programs containing up to 128 symbols. Programs containing up to 1024 symbolic labels may be assembled if the maximum possible 8K of memory is allocated to the symbol table. Refer to appendix e. for details. Programs which require more than 1024 symbols, will need to be independently assembled in sections.

As a guide, a program with 8K of source code will normally contain about 128 symbolic labels and produce about 1K of object code. This will cover the majority of programs to be assembled.

9.9 SYMBOLIC LABELS

The assembler allows symbolic names to be any reasonable length. However only the first six characters of a symbolic name is stored by the assembler. This means that the first six characters of each label must be unique to avoid duplicating a label.

Since the source listing format is designed to accommodate labels which are a maximum of six characters long, it is recommended that labels should be limited to a length of six characters or less.

Section 3. - APPENDICES

Appendix a. EDITOR SYNTAX

A/string/	APPEND string to the end of the current line.
AS	Assemble the file currently open in memory.
B,n	Move the current line pointer BACK n lines and list the line.
C/old/new/,n	Search the next n lines from the current line and CHANGE the first occurrence of 'old' in each line to 'new'.
C/old/new/,n*	CHANGE ALL occurrences on each line. n must be less than 256.
CO,n,<address>	COPY n lines, including the current line and append them to the file starting at the HEX address specified. A new file is opened if none exists.
D,n	DELETE n lines including the current line.
E,<address>	EXIT from the editor and Jump to the HEX address specified. If no address is specified, exit is made via the branch in the personality module.
F/text/	FIND the first occurrence of the string 'text' from the next line onwards. If the string is not found the current line pointer is not moved.
G,n	GO to the nth line in the file and list it.
G,<address>,n	GO to the address and list 'n' lines.
I	Enter the 'insert text mode'. This is indicated by a '-' as the prompt symbol. Text is inserted before the current line. Exit from the insert text mode by entering a null line.
L,n	LIST the next 'n' lines including the current line. The current line pointer is not moved.
LO	LOAD a source file. The format of the command depends on the personality module.
M,<address>	MERGE the contents of the secondary file starting at the specified HEX address into the primary file BEFORE the current line.

N,n	Move the current line pointer forward over the NEXT n lines and list the line.
NE.<address>	Open a NEW file starting at the address specified. Default address is 2000.
O.<address>	Check for a valid file at the specified address and open this OLD file for use if present. The default address is 2000.
S	LIST addresses of start of text, current line and end of text of the currently open file.
SA	SAVE the file currently open in memory onto tape. The precise format will depend on the personality module installed.
T	Move the current line pointer to the TOP of the file.
X	Enter the line edit mode. LF FORWARD one position BS BACKWARDS one position DEL DELETE character under cursor CTL X DELETE to end of line
null	Set pointer to next line and list it.

',n' may be replaced by '*' to mean ALL. eg. L* will list the whole file, from the current line to the end of the file. If ',n' is omitted, a default of 1 is assumed.

Appendix b. EDITOR ERROR MESSAGES

1	* *	No primary file. A file has not yet been opened.
2	*N*	File or String not found.
3	*I*	Illegal file (bit 7 set on a character in file)
4	*O*	Input buffer overflow. Line too long.
5	*?*	Syntax error. Command not understood.
6	*E*	Bottom of file reached.
7	*S*	Top of file reached.
8	*L*	Locked file. ROM or no memory.
9	*M*	Memory overflow error.

Appendix c. ASSEMBLER ERROR CODES

B	Branching error. Relative branch out of range.
D	Duplicate label.
F	Full symbol table. Too many symbols.
I	Indexing error. Index register greater than 3.
L	Label error. Label does not begin with a letter.
M	Mnemonic error. Instruction not valid.
O	Operand error. Operand does not evaluate within range.
P	Pasing error. Illegal reference outside current page.
R	Register error. Register greater than 3 specified.
S	Syntax error.
U	Undefined symbol. Reference to a symbol not defined.

Appendix d. ASSEMBLER SYNTAX

This assembler uses the same format as detailed in the SIGNETICS 2650 PROGRAMMING MANUAL with the following exceptions and clarifications.

1. The following PSEUDO INSTRUCTIONS are supported:-

	ORG	n	Set program counter to n.
Label	EQU	expr	Assign value of expr to label.
	RES	n	Reserve n bytes of memory. n<256.
	ACON	n	Form a double byte address constant n.
	DATA	n,m	Create data bytes. Decimal, Hex and ASCII allowed.
	DFLT	1	Set default data type to Hexadecimal.
	PAG		Start new page of listing.
	SPAC	n	Leave n blank lines in listing.
	END	expr	End assembly. Only required at end of the final pass. Expr is an execute address used when an object tape is produced.

2. LODZ,R1 should be used instead of LODZ R1 for all resister to resister instructions.

3. Auto increment or auto decrement is specified as SPC,R2+ not SPC,R2,+. There is no comma between the resister and the plus or minus sign, as specified in the Signetics manual.

4. '>' , meaning 'take the lower byte of a 16 bit quantity', is not supported, as the assembler will do this by default . '<' is supported and implies 'take the upper byte of a 16 bit quantity'.

5. The source file MUST contain at least one ORG statement and this must occur before any assembled output would be generated. There is no default value for the origin.

6. An END statement should be placed at the conclusion of the LAST segment of a source file to ensure that an 'end of tape block' will be generated and the printer will form feed at the end of the assembly. Otherwise the END statement is optional.

Appendix e. ADAPTING TO YOUR SYSTEM

TERMINAL INPUT/OUTPUT ROUTINES

This program should function without any changes being required if it is run on a computer which uses PIPBUG or BINBUG as a monitor, and which has continuous read/write memory installed between 400 and 2FFF. If the character input and character output routines in the monitor being used do not have entry points at 0286 and 02B4 respectively, then it will be necessary to change the Jumps at 0609 and 060C to point to the start of these subroutines in the monitor used.

These I/O subroutines must emulate the PIPBUG subroutines.

CHIN (609) returns the character input in R0.

COUT (60C) outputs the character in R0 to the terminal.

Each of these subroutines must:

- preserve R1, R2, R3.
- return with the lower registers selected.
- not nest subroutines deeper than three levels.

EXIT (606) This branches to the monitor entry point (22), when 'E', with no address specified, is used to exit from the editor. If your monitor does not have its re-entry point at 22, this Jump will need to be altered.

TSTBRK (60F) Whenever a character is output by the program either to the terminal or to the printer, a call is made to the subroutine TSTBRK to test for a break from the keyboard. Currently this tests the sense input. If a BREAK has occurred, this routine should return with the Condition Code set NOT EQUAL.

PRINTER OUTPUT ROUTINES

In order to output to a printer from this program, it is necessary to provide the appropriate driver routines.

OUT2 (612) must output the character in R0 to the printer following the same rules as for COUT.

INIT (615) This routine may not be required by all printers, however the branch is provided to allow the printer to be turned on and initialised if this is required.

TERMP (618) This routine may not be required, however it is provided to allow the printer the be turned off if required.

PLNG (637) contains the number of lines of assembler output which will be listed on each page. As supplied, this is 55 (37 hex) which results in a page length of 66 lines including header and footer. This may be changed to give a different page length.

FTST (639) is the string of characters which is output at the bottom of each page. Currently this is four linefeeds and a cut mark, but may be replaced with a Form Feed if your printer responds to this character.

If your printer requires the output to pause at the end of each page so that a new sheet of paper may be inserted, this may be implemented by writing a printer driver routine which detects the formfeed character and then waits for a character to be input from the keyboard before continuing.

STORAGE MEMORY ALLOCATION

TEXT BUFFER

As supplied, the text buffer begins at 2000 and ends at 2FFF (4K). If your system contains more memory, these limits may be altered by changing the ACONs DEFL (61C) and MYND (61E) in the personality module. DEFL points to the first memory location used by the editor for its default text buffer. MYND points to the last location available for use by the text buffer. The text buffer may extend across 8K page boundaries.

SYMBOL TABLE

The assembler requires an area of memory in which to store its table of symbols. The amount of memory required will depend on the number of symbols used. Each symbol requires 8 bytes and the maximum symbol table size is 8K, because it cannot extend across a page boundary. As supplied, the symbol table is located between 1C00 and 1FFF which will allow 128 symbols to be stored. If more symbols are required the symbol table will need to be moved. The start of the symbol table is pointed to by STAB (620) and the end by STABND (622). Provided memory is available, the symbol table could be moved to page 2 by altering STAB (61C) from 1C00 to 4000 and STABND (61E) from 1FFF to 5FFF. This will allocate the whole of page 2 to the symbol table and allow up to 1024 symbols to be used.

Take care that the text and symbol table do not overlap.

SAVE AND LOAD

Pointers are provided in the personality module to allow custom tape SAVE and LOAD routines to be grafted onto the editor.

Each of these routines must be a subroutine and set up its pointers to the start and end of the text file from those of the editor.

Users intending to undertake this task are advised to study a source listing carefully to determine how to achieve this.

ASSEMBLER OBJECT TAPE ROUTINES

Provision is made to patch to user provided object tape output routines. The assembler fills a buffer with object code until the buffer is full, or the program counter is not contiguous between output bytes, and then branches to a subroutine via an ACON to output the buffer to tape. The following ACONS are provided and may point anywhere in memory.

BUFS (630) Points to the start of the tape buffer.

BUFP (632) Points to the location used as the counter for the number of bytes in the buffer.

TOPC (634) Points to the locations used to store the address in memory of the first byte in the buffer.

BUFE (636) Contains the maximum length of the buffer. This must be between 1 and 256. 0 implies 256.

The assembler will branch through three ACONS at appropriate times to the following routines.

TINT (62A) This Subroutine should initialise the cassette system. It should start the recorder and put out a leader. If no initialisation is required, this ACON should point to a subroutine return statement.

DTAP (628) This subroutine must output to tape in the required format, the number of bytes indicated by the location pointed to by BUFP. The bytes to be output are contained in a buffer starting at the address pointed to by BUFS. The storage address for the first object byte is contained in the bytes pointed to by TOPC.

TEND (62C) This subroutine must output to tape an end of tape block and turn off the recorder.

Any of the 2650's registers may be used by these routines, but each must return with the lower resister bank selected and must not nest subroutines deeper than three.

Appendix f. DEFAULT DATA TYPE

As supplied, the assembler will interpret data statements which are not otherwise specified, as DECIMAL numbers. This may be changed so that the default data type is HEXADECIMAL by making the DATA TYPE SWITCH, DARGF (628) non-zero.

```
DARGF  =  0    Default data type is DECIMAL
DARGF  = FF    Default data type is HEXADECIMAL.
```

The default data type may be switched during assembly with the pseudo-instruction DFLT.

```
        DFLT    0       force DECIMAL default option
        DFLT    1       force HEXADECIMAL default option
```

A hexadecimal number which starts with the letters A-F must be preceded by 0 to distinguish it from a symbolic label.

```
12      0A7     0FFFF   are valid hex constants
```

Appendix 5. A SAMPLE ASSEMBLY

OBJECTIVE To assemble the following test program, which contains two errors, and correct the errors using the editor.

```
* TEST PROGRAM TO DEMONSTRATE ASSEMBLER
*
       ORG     H'440'  Program starts here in memory
*
COUT   EQU     H'2B4'  Monitor output subroutine
*
START  LODI,R3 20      Set up counter
       LODI,R0 A'U'    Character to be output
LOOP   BSTA,UN COUT    Output it
       BDRR,R3 LOP     Do it 20 times
       BCTA,UN MBUG    Exit to monitor
       END     START   Execute address
```

Firstly , load the Editor/Assembler program from tape.

Execute the program by typing G600.

The editor's prompt '>' will be displayed. The cursor should be over the prompt.

Type 'NE' to open a NEW file.

Type 'I' and return to enter the INSERT TEXT mode. The prompt will now be '-' to indicate that text is being entered into the file.

Type in the program listed above. Use TAB characters to format the program into columns. Spaces could be used but these will use more memory.

Once the program has been typed in, assemble it by entering the command AS followed by a return.

OPTION?

will be displayed on the terminal. Respond to this with a return.

The assembler will assemble the file and list any lines containing errors, but will produce no assembler output, either machine code or source listing. In this case

```
U  0006      :        BDRR,R3 LOP
U  0007      :        BCTA,UN MBUG
02 ERRORS DETECTED
```

will be displayed, indicating that two errors have been detected and these are the two lines in error. In both cases the error occurred because neither label was defined. LOP is a typing error and should have been LOOP, while MBUG must be defined using an EQU pseudo-instruction.

After assembly control is returned to the editor. Line 6 may be corrected by typing

```
G,6                     return
```

to access line 6, which will cause it to be displayed on the terminal.

```
C/LOP/LOOP/             return
```

will correct this spelling mistake.

```
T                       return
```

will return the current line pointer to the top of the file.

```
I                       return
-MBUG   EQU    H'22'    return
-                       return
>
```

will define the label MBUG, and make the source code ready for another attempt at assembly. Again type

```
AS                      return
OPTION?                 return
```

This time you should get

```
00 ERRORS DETECTED
```

displayed indicating that the assembler has found no syntax errors.

To display a source listing on the screen, type in

```
AS                      return
OPTION? L               return
HEADING? TEST HEADING   return
OPTION?                 return
```

a formatted listing of the object code and assembler code will be displayed on the terminal.

To output this source listing to a printer type

```
AS                      return
OPTION? P               return
HEADING? TEST HEADING   return
```

```
OPTION?                    return
```

Note: A subroutine to output characters to a printer must have been provided or this option will produce no output.

To store the object code in memory, type

```
AS                         return
OPTION? M                  return
OPTION?                    return
```

Then to exit from the editor and execute the program type

```
E.440
```

(440 is where the program starts). This should cause a line of 20 'U's to be printed on the terminal.

CAUTION: Be very careful not to assemble directly into memory, code which will overwrite the Editor/Assembler program, the program's source file, or the symbol table area.

To output the object file to tape, type

```
AS                         return
OPTION? T                  return
OPTION?                    return
```

The address of the label START will be recorded on the tape as an execute address.

Any of these options may be used together.

Appendix h. MEMORY MAP

450 - 5FF Editor and assembler scratch

460 - 55F Tape output and editor input buffer
570 Start of file pointer
572 Current line pointer
574 End of file pointer

600 - 66C Personality module constants and jumps

600 Hard start
603 Soft start
606 Program exit to monitor
609 Keyboard input routine
60C Terminal output routine
60F Test for BREAK
612 Printer output jump
615 Printer intitialise jump
61C Default start of file pointer
61E End of text buffer
620 Start of symbol table pointer
622 End of symbol table storage

Tape routine pointers
624 Load source file from tape
626 Save source file on tape
628 Dump object block to tape
62A Initialise tape system
62C Output last object block to tape
62E Load next source block into memory
630 Start of tape buffer
632 Location of tape buffer pointer
634 Location of object block address
636 Length of tape buffer

637 Lines per page, less headers
638 Default argument switch
639 - 640 Footer strins
641 - 643 Listins TAB stops
644 - 66F Messase strinss

670 - 1BFF Editor / Assembler program

1C00 - 1FFF Symbol table storage *

2000 - 2FFF Text Buffer *

* The text buffer and symbol table storage are movable. See appendix e. for details.