UniFLEX™ 6809 Assembler

Technical Systems Consultants, Inc.

UniFLEX™ 6809 Assembler

MANUAL REVISION HISTORY

Revision	Date	Change
A	8/80	Original Release
B	4/81	Corrected "base" example; Added 2 previously omitted error messages
C	7/82	Added documentation of "m" option.

DISCLAIMER

The supplied software is intended for use only as described in this manual. Use of undocumented features or parameters may cause unpredictable results for which Technical Systems Consultants, Inc. cannot assume responsibility. Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Technical Systems Consultants, Inc. will not assume responsibility for any damages incurred or generated by such material. Technical Systems Consultants, Inc. reserves the right to make changes in such material at any time without notice.

Table of Contents

I. INTRODUCTION

This manual describes the operation and use of the UniFLEX™ 6809 Assembler. The assembler accepts all standard Motorola mnemonics for the 6809 instruction set as well as all standard 6800 and 6801 mnemonics. Macros and conditional assembly are supported as well as numerous other directives for convenient assembler control. The assembler executes in two passes and can accept any size file so long as sufficient memory is installed to contain the symbol table. Output is in the form of a binary disk file as well as an assembled listing output which may be routed to a printer or to a disk file through the facilities of UniFLEX.

This manual is by no means intended to teach the reader assembly language programming nor even the full details of the 6809 instruction set. It assumes the user has a working knowledge of assembly language programming and a manual describing the 6809 instruction set and addressing modes in full. The former can be acquired through any of a large number of books available on assembly programming, the latter from the 6809 hardware manufacturer or seller.

Throughout the manual a couple of notational conventions are used which are explained here. The first is the use of angle brackets ('<' and '>'). These are often used to enclose the description of a particular item. This item might be anything from a filename to a macro parameter. It is enclosed in angle brackets to show that it is a single item even though the description may require several words. The second notation is the use of square brackets ('[' and ']'). These are used to enclose an optional item.

™ UniFLEX is a trademark of Technical Systems Consultants, Inc.

II. GETTING THE SYSTEM STARTED

The UniFLEX 6809 Asssembler is very simple to operate. There are no built-in editing functions - you must have a previously edited source file on disk before using the assembler. This file must be a standard UniFLEX text file which is simply textual lines terminated by a carriage return. There should be no line numbers or control characters (except for the carriage returns and tabs) in the file. When you have both the assembler and the edited source file on a disk or disks which are inserted in a powered up system, you are ready to begin.

The Command Line

The very minimum command line necessary to execute an assembly is as follows:

```
++ asmb sourcefile
```

The plus signs are UniFLEX's ready prompt, "asmb" is the name of the assembler command file, and the "sourcefile" is a standard UniFLEX specification for the source file you wish to assemble.

As stated, this is the very minimum command which can be used. It is possible to supply many more parameters or options to the assembler, but if left off as in this example, the assembler will assume default parameters. Perhaps the most important options available are those associated with output. There are two types of output available from the assembler: object code output and assembled source listing output. The options regarding the assembled source listing output will be described a little later.

Object code is produced into a UniFLEX binary file. That file may be absolute or "segmented" as explained in the section on Object Code Production. It is also possible to disable production of the object code file. Since no specifications are made concerning object code output in the above example, the assembler will assume the default case which is to produce a binary disk file. Since no name was specified, the output binary file will assume the same name as the input source file specified but with the characters ".b" appended. If there is not room to append those two characters, the last one or two characters of the input file name will be truncated to make room. In our above example, the created binary file would be named "sourcefile.b". Should a file exist with the same name, it will be automatically deleted with no prompting.

If you wish to create a binary file by another name, you may do so by placing the desired file specification on the command line as follows:

```
++ asmb sourcefile +o=binaryfile
```

The "+o=" is an option to the assembler which specifies that an Output file is being created. This example would result in a binary file named "binaryfile". Again, if a file by that name already existed, it would be deleted to permit creation of the new binary file.

Multiple Input Source Files

The UniFLEX 6809 Assembler is capable of accepting more than one file as the source for assembly. If multiple input files are specified, they are read in the calling order and assembled together to produce a single output file. This permits the user to break source programs down into more convenient size source files which may then be assembled into one binary file. As mentioned, the files are read sequentially in the calling order with the last line of source from the current file being followed immediately by the first line of the ensuing file. All "end" statements in the source are effectively ignored and the assembly is terminated when the last line of the last source file is read.

There are two ways to specify multiple input files to the assembler, by entering the name of each file and by a match list in a file specification. Entering each filename would look like this:

```
++ asmb file1 file2 file3 file4
```

Using a match list in the file specification we might have:

```
++ asmb file[1-4]
```

In this example, the square brackets do not denote an optional item, but rather are the method of specifying a list of match characters under UniFLEX. Both of the above examples would produce the same result. Note that in these examples a binary file would be created by default and would be called "file1.b" (the name is taken from the first input file). As before we could specify a binary file name as follows:

```
++ asmb file1 file2 file3 file4 +o=command
```

which would result in a binary file called "command".

Specifying Assembly Options

Now we shall go one step further and add a set of single character option flags which may be set on the command line as follows:

```
++ asmb sourcefile +options
```

The plus sign is required to separate the option(s) from the file specification(s). In this example, the word "options" following the plus sign represents a single character option flag or list of character option flags which either enable or disable a particular option or options. In all cases they reverse the sense of the particular option from its default sense. There may be any number of options specified and they may be specified in any order. There may not be spaces within the option list. Following is a list of the available options and what they represent:

+b Do not create a binary file on the disk. No binary file will be created even if a binary file name is specified. This is useful when assembling a program to check for errors before the final program is completed or when obtaining a printed source listing.

+l Suppress the assembled listing output. If not specified, the assembler will output each line as it is assembled in pass 2, honoring the 'lis' and 'nol' options (see the 'opt' directive). Those lines containing errors will always be printed regardless of whether or not this option is specified.

+s Suppress the symbol table output. The assembler normally prints out a sorted symbol table at the end of an assembly. This option suppresses that output. Note that the 'l' option will not suppress the symbol table output, just the source listing itself.

+g Turns on printing of multiple line object code instructions. This allows certain directives (fcb, fdb, and fcc) to produce several lines of output listing for only one instruction line. Without this option the first line of output from such an instruction (the line which contains the source) is printed but subsequent lines which contain only object code information are suppressed.

+n Enables the printing of decimal line numbers on each output line. These numbers are the consecutive number of the line as read by the assembler. Error lines are always output with the line number regardless of the state of this option.

+d Suppress printing of the date in the header at the top of each output page. The assembler normally picks up the current date from UniFLEX and prints it in the header. This option causes the date to be omitted.

+a Causes the absolute address of all relative branches to be printed in the object code field of the assembled output listing.

+w Suppress warning messages. The 6809 assembler is capable of reporting a number of warning messages as well as an indicator of long jumps and branches that could be shortened. This option suppresses the printing of these messages and indicators.

+f Disables the auto-fielding feature of the assembler such that assembled output lines appear in the exact form as found in the input file.

+0000 This option is different from those above in that it is not a character, but a decimal number. The number specifies the approximate number of symbol table entries required for the assembly. The assembler defaults to about 700 entries, but with this option the user can specify up to 3000 entries. If placed in a list with other options, this number option must be the last one.

The following options are slightly different than those listed above in that they require a parameter to be supplied. This parameter is supplied by following the option character with an equal sign and then the parameter. Since the parameter is terminated by a normal UniFLEX terminator character (such as a space), only one of these type options may be specified per plus sign (they may not be included in a list). More than one can be supplied in a command by using another plus sign.

+m=00 Specify amount of memory reserved for macros. Amount is expressed in Kilobytes and may range from 2 to 12. Default amount is 2K.

+o=file Allows specification of an output binary file name (in this example "file"). This option has been previously described.

+p=0000 Permits starting assembled output listing with a specified page number. Here that page number is represented by "0000". For details of options see following text.

+a=prm1
+b=prm2
+c=prm3 These options permit the specification of three "command line parameters" named "a", "b", and "c". In this case the parameters supplied are "prm1", "prm2", and "prm3". Details of command line parameter specification follow.

Specifying a Starting Page Number

As mentioned in the 'p' option above, it is possible to specify a particular page number at which printing of the assembled listing should commence. All output before that page is suppressed, INCLUDING ERROR LINES! When the specified page is hit, printing begins and continues to the end of the assembly. Note that it is possible to suspend or to completely terminate an assembly during output by use of the escape or 'ctrl c' special characters found in UniFLEX. The page number is specified as shown above and may be any decimal number from 1 to 65535. The page number itself must be directly followed by a UniFLEX terminator such as a space. It is also important to note that the 'pag' mode must be selected (pagination turned on) in order for the 'p' option to have any effect. The 'pag' mode is on by default or may be turned on as described in the section on Standard Directives.

Command Line Parameters

The assembler has a facility for passing information from the command line directly into the source program. A maximum of three pieces of information or "command line parameters" may be passed to the program. These parameter are simply strings of characters that will be substituted into the source listing as it is read in by the assembler. These parameters are expressed in the command line as shown above. The string of characters which makes up an individual command line parameter cannot have spaces or commas imbedded in it. For further information on command line parameters and how to specify where in the source program these parameters should be substituted, see the section on Special Features.

Order for Specifying Filenames, Options, and Parameters

Input filenames, options, and command line parameters can be specified to the assembler in any order. The assembler scans the input command line twice, once to pick out all options and parameters (they all begin with a plus sign) and then again to pick out all file specifications. The only place order is significant is when multiple input files are specified. They will be assembled in the order entered on the calling line.

Outputting to a Hardcopy Device

The assembler does not have a built in means for outputting the assembled listing to a hardcopy device. This operation is, however, available through the facilities of UniFLEX. The most common means are to route the standard output to a file which may later be spooled or to pipe the standard output to a spooler.

EXAMPLES:

++ asmb test

Assembles a file called "test" and creates a binary file called "test.b" in the same directory. The assembled listing is output to the terminal as is the symbol table.

++ asmb test +ls

Same as before except that no listing is output (except for any lines with errors) and no symbol table is output.

++ asmb test +o=/bin/test +ls

Assembles a file in the current directory called "test" and produces a binary file in the "bin" directory called "test". No listing or symbol table is output and if a file by the name of "test" already resides in the "bin" directory, it will be automatically deleted before the assembly starts.

++ asmb /usr/john/main +bngs

This command assembles the file "main" in John's directory but does not produce a binary file. The assembled listing is output with line numbers and multiple line generated code turned on. No symbol table is printed.

++ asmb file[1-4] +bsn +p=26

This command assembles all files beginning with "file" and ending with a 1, 2, 3, or 4. No binary nor symbol table is output and line numbers are turned on. The assembled listing output does not begin until the assembler reaches page number 26.

++ asmb +ga dumper +a=mini +b=125 +n

This command demonstrates command line parameters and the fact that the filenames, parameters, and options can come in any desired order on the command line. The assembled listing is output with multiple line code generation and absolute address printing for relative branches both enabled. The file to be assembled is called "dumper". There are two command line parameters which may be passed into the source listing. The first is the single word "mini" and the second is the string "125". Finally, line numbers are enabled for the output listing.

III. ASSEMBLER OPERATION & SOURCE LINE COMPONENTS

The TSC Assembler is a 2 pass assembler. During pass one a symbolic reference table is constructed and in pass two the code is actually assembled, printing a listing and outputting object code if desired. The source may be supplied in free format as described below. Each line of source consists of the actual source statement terminated with a carriage return (0D hex). The source must be comprised of ASCII characters with their parity or 8th bit cleared to zero. Special meaning is attached to many of these characters as will be described later. Control characters (00 to FF hex) other than the carriage return (0DH) and the horizontal tab (09H) are prohibited from being in the actual source statement part of the line. Their inclusion in the source statement will produce undefined results. Each source line is comprised of up to four fields: Label, Opcode, Operand, and Comment. With two exceptions, every line must have an opcode while the other fields may or may not be optional. These two exceptions are:

1) "Comment Lines" may be inserted anywhere in the source and are ignored by the assembler during object code production. Comment lines may be either of two types:

   a) Any line beginning with an asterisk (hex 2A) in column one.
   b) A null line or a line containing only a carriage return. While this line can contain no text, it is still considered a comment line as it causes a space in the output listing.

2) Lines which contain a label but no opcode or operand field.

SOURCE STATEMENT FIELDS

The four fields are described here along with their format specifications. The fields are free format which means there may be any number of spaces separating each field. In general, no spaces are allowed within a field.

LABEL OR SYMBOL FIELD:

This field may contain a symbolic label or name which is assigned the instruction's address and may be called upon throughout the source program.

1) The label must begin in column one and must be unique. Labels are optional. If the label is to be omitted, the first character of the line must be a space.
2) A label may consist of letters (A-Z or a-z), numbers (0-9), or an underscore (_ or 5F hex). Note that upper and lower case letters are not considered equivalent. Thus 'ABC' is a different label from 'Abc'.
3) Every label must begin with a letter or underscore.
4) Labels may be of any length, but only the first 6 characters are significant.
5) The label field must be terminated by a space, tab, or a return.

OPCODE FIELD:

This field contains the 6809 opcode (mnemonic) or pseudo-op. It specifies the operation that is to be performed. The pseudo-ops recognized by this assembler are described later in this manual.

1) The opcode is made up of letters (A-Z or a-z) and numbers (0-9). In this field, upper and lower case may be used interchangably.
2) This field must be terminated by a space or tab if there is an operand or by a space, tab, or return if there is no operand.

OPERAND FIELD:

The operand provides any data or address information which may be required by the opcode. This field may or may not be required, depending on the opcode. Operands are generally combinations of register specifications and mathematical expressions which can include constants, symbols, ASCII literals, etc. as explained later.

1) The operand field can contain no spaces or tabs.
2) This field is terminated with a space, tab, or return.
3) Any of several types of data may make up the operand: register specifications, numeric constants, symbols, ASCII literals, and the special PC designator.

COMMENT FIELD:

The comment field may be used to insert comments on each line of source. Comments are for the programmer's convenience only and are ignored by the assembler.

1) The comment field is always optional.
2) This field must be preceded by a space or tab.
3) Comments may contain any characters from SPACE (hex 20) thru DELETE (hex 7F) and the tab character.
4) This field is terminated by a carriage return.

REGISTER SPECIFICATION

Many opcodes require that the operand following them specify one or more registers. EXG and TFR require two registers specified, push and pull allow any number, and the indexed addressing mode requires specification of the register by which indexing is to be done. The following are possible register names:

A,B,CC,DP,X,Y,U,S,D,PC

The EXG and TFR instructions require two register specs separated by a comma. The push and pull instructions allow any number of registers to be specified, again, separated by commas. Indexed addressing requires one of X,Y,U, or S as explained under the indexed addressing mode description.

EXPRESSIONS

Many opcodes require that the operand supply further data or information in the form of an expression. This expression may be one or more items combined by any of four operator types: arithmetic, logical, relational, and shift.

Expressions are always evaluated as full 16 bit operations. If the result of the operation is to be only 8 bits, the assembler truncates the upper half. If truncation occurs when warnings are enabled, an appropriate message will be issued.

No spaces or tabs may be imbedded in an expression.

ITEM TYPES:

The "item or items" used in an expression may be any of four types as listed below. These may stand alone or may be intermixed by the use of the operators.

1) NUMERICAL CONSTANTS: Numbers may be supplied to the assembler in any of the four number bases shown below. The number given will be converted to 16 bits truncating any numbers greater than that. If 8 bit numbers are required, the 16 bit number will then be further truncated to 8 bits with notification of such if warning messages are enabled. To specify which number base is desired, the programmer must supply a prefix character to a number as detailed below.

BASE	PREFIX	CHARACTERS ALLOWED
Decimal	none	0 thru 9
Binary	%	0 or 1
Octal	@	0 thru 7
Hexadecimal	$	0 thru 9, A thru F

If no prefix is assigned, the assembler assumes the number to be decimal.

2) ASCII CONSTANTS: The binary equivalent of a single ASCII printable character may be supplied to the assembler by preceding it with a single quote. The character should be between 20 and 7F hex. The binary equivalent of two ASCII printable characters (the first in the upper 8 bits and the second in the lower 8 bits) may be supplied by preceding the two characters with a double quote.
3) LABELS: Labels which have been assigned some address or constant value may be used in expressions. As described above under the label field, a label is comprised of letters, digits, and hyphens beginning with a letter. The label may be of any length, but only the first 6 characters are significant. Any label used in the operand field must be defined elsewhere in the program. None of the standard 6809 register specifications should be used as a label.
4) PC DESIGNATOR: The asterisk (*) has been set aside as a special PC designator (Program Counter). It may be used in an expression just as any other value and is equal to the address of the current instruction.

EXPRESSION OPERATORS

As mentioned previously, the four classes of operators are: arithmetic, logical, relational, and shift. These operators permit assembly-time operations such as addition or division to take place. "Assembly-time" means that the expression is evaluated during the assembly and the result becomes a permanent part of your program.

a) ARITHMETIC OPERATORS
The arithmetic operators are as follows:

Operator	Meaning
+	Unary or binary addition
-	Unary or binary subtraction
*	Multiplication
/	Division (any remainder is discarded)

b) LOGICAL OPERATORS
The logical operators are as follows:

Operator	Meaning
&	Logical AND operator
|	Logical OR operator
!	Logical NOT operator
>>	Shift right operator
<<	Shift left operator

The logical operations are full 16 bit operations. In other words for the AND operation, every bit from the first operand or item is individually AND'ed with its corresponding bit from the second operand or item. The shift operators shift the left term the number of places indicated by the right term. Zeroes are shifted in and bits shifted out are lost.

c) RELATIONAL OPERATORS
The relational operators are as follows:

Operator	Meaning
=	Equal
<	Less than
>	Greater than
<>	Not equal
<=	Less than or equal
>=	Greater than or equal

The relational operations yield a true-false result. If the evaluation of the relation is true, the resulting value be all ones. If false, the resulting value will will be all zeros. Relational operations are generally used in conjunction with conditional assembly as shown in that section.

OPERATOR PRECEDENCE

Certain operators take precedence over others in an expression. This precedence can be overcome by use of parentheses. If there is more than one operator of the same priority level and no parentheses to indicate the order in which they should be evaluated, then the operations are carried out in a left to right order.

The following list classifies the operators in order of precedence (highest priority first):

1) Parenthesized expressions
2) Unary + and -
3) Shift operators
4) Multiply and Divide
5) Binary Addition and Subtraction
6) Relational Operators
7) Logical NOT Operator
8) Logical AND and OR Operators

IV. The INSTRUCTION SET

This section is a quick introduction to the 6809 architecture and insruction set. It is by no means complete. The intention is to familiarize the user who is already proficient at 6800 assembly language programming with the basic structure of 6809 assembly language. For more complete details on the 6809 instruction set you should obtain the proper documentation from the hardware manufacturer.

Programming Model

The 6809 microprocessor has 9 registers that are accessible by the programmer. Four of these are 8-bit registers while the other five are 16-bit registers. Two of the 8-bit registers can, in some instances, be referenced as one 16-bit registers. The registers are as follows:

The 'A' accumulator (A)	8 bit
The 'B' accumulator (B)	8 bit
The Condition Code register (CC)	8 bit
The Direct Page register (DP)	8 bit
The 'X' index register (X)	16 bit
The 'Y' index register (Y)	16 bit
The User stack pointer (U)	16 bit
The System stack pointer (S)	16 bit
The Program Counter (PC)	16 bit

The A and B accumulators can often be referenced as one 16-bit register represented by a 'D' (for Double-accumulator). In these cases, the A accumulator is the most significant half.

The Addressing Modes

There are several possible addressing modes in the 6809 instruction set. One of the best features of the 6809 is the consistency or regularity built into the instruction set. For the most part, any instruction which addresses memory can use any of the addressing modes available. It is not necessary to remember which instructions can use which addressing modes, etc. The addressing modes and a brief description of each follow.

1) Inherent
   Inherent addressing refers to those instructions which have no addressing associated with them.
   Example: ABX add B accumulator to X

2) Accumulator
Accumulator addressing is done in those instructions which can specify the A or B accumulator. In some cases this may be the 16-bit D accumulator.
Example: DECA decrement the A accumulator

3) Immediate
In Immediate addressing the byte or bytes following the opcode are the information being addressed. These byte or bytes are specified as part of the instruction.
Example: LDA #8 load immediate value (8) into A

4) Relative - Long and Short
In Relative addressing, the value of the byte(s) immediately following the opcode (1 if short, 2 if long) are added as a two's complement number to the current value of the program counter (PC register) to produce a new PC location. In the source code, the programmer specifies the desired address to which execution should be transferred and the assembler determines the correct offset to place after the opcode.
Example: LBRA THERE the program will branch to THERE

5) Extended
In Extended addressing, the two bytes (16-bits) following the opcode are used as an absolute memory address value.
Example: LDA $1000 load A from memory location 1000 hex

6) Direct
In Direct addressing, the single byte (8-bits) following the opcode is used as a pointer into a 256-byte window or "page" of memory. The page used for this purpose is the one currently found in the Direct Page register. Thus, the effective address is a concatenation of the Direct Page register as the most significant half and the byte following the opcode as the least significant half.
Example: LDA $22 load A from memory location $XX22 where XX represents the contents of the DP register

7) Extended Indirect
In Extended Indirect addressing, the 16-bit value following the opcode is used to point to two bytes in memory which are used as the effective address.
Example: LDA [$A012] loads A from the address stored at locations $A012 and $A013

8) Indexed

The Indexed addressing mode of the 6809 is an extremely powerful method of specifying addresses which is, in general, some sort of offset from the value stored in one of the registers X, Y, U, S, or PC. There are several forms of indexed addressing which could each be considered an addressing mode in itself. We will, however, discuss each as a subset of Indexed addressing in general. Note that except for the Auto-increment and Auto-decrement modes, determining the effective address has no effect on the register being used as the index.

8a) Constant-Offset Indexed

This mode uses a two's complement offset value found in the byte or bytes following the opcode. The offset is added to the contents of the specified register to produce a 16-bit effective address. The offset may be represented as a number, a symbol, or any valid expression. It can be either positive or negative and can be a full 16-bits.

```
Example:  LDA 0,X   loads A from location pointed to by X
          LDA 5216,Y  loads A from (Y) plus 5216
          LDA -36,U  loads A from (U) minus 36
          LDA VAL,S  loads A from (S) plus VAL
```

8b) Accumulator Indexed

In Accumulator indexing, the contents of the specified accumulator (A, B, or D) are added to the specified indexing register as a two's complement value. The result is the effective address.

```
Example:  LDA B,Y  loads A from (B)+(Y)
          LDX D,S  loads X from (D)+(S)
```

8c) Auto-Increment

The contents of the selected register are used as the effective address with no offset permitted. After that effective address has been determined, the selected register is incremented by one (for single plus sign) or two (double plus sign).

```
Example:  LDA 0,X+  loads A from X then bumps X by 1
          LDD ,Y++  loads D from Y then bumps Y by 2
```

8d) Auto-Decrement

In auto-decrementing, the selected register is first decremented by one (single minus sign) or two (double minus sign). The resulting value, with no offset, is used as the effective address.

```
Example:  LDA 0,-U  decrements U by 1 then loads A from address in U
          LDU ,--S  decrements S by 2 then loads U from address in S
```

Further Addressing Modes

Indexed Indirect Addressing

All the Indexed Addressing modes above can also be used in an indirect fashion by enclosing the operand in square brackets. When this is done, the effective address as described in all the above modes is no longer the final effective address. Instead, the two bytes pointed to by that address are used as the effective address.

```
Examples:  LDA [,X]   loads A from the address pointed to by X
           LDX [D,U]  loads X from the address pointed
                      to by (U)+(D)
```

If auto-increment or auto-decrement addressing is done in an indirect fashion, they must be a double increment (two plus signs) or double decrement (two minus signs).

PC Relative Addressing

Indexing may be done from the PC register just as from the X, Y, U, or S. The general use of indexing from the PC register is to address some value in a position-independent manner. Thus if we address some value at the current PC plus 10, no matter where the program executes the value will always be addressed. The programmer does not usually know what that constant offset should be, he knows the address of the value he wants to access as an absolute value for the program as assembled. Thus a mechanism has been included in the assembler to automatically determine the offset from the current PC to that absolute address. This mechanism is called PC Relative Addressing. The value specified in a PC Relative address operand is the absolute value. The assembler takes the difference between this absolute value and the current PC and generates that offset as part of the assembled code for the instruction. PC Relative Addressing is distinguished from normal PC Offset Indexing by the use of 'PCR' as the register name instead of 'PC'.

```
Example:  LEAX STRING,PCR
                this instruction determines the offset
                between the PC and STRING and uses it
                as an offset for the PC register to
                determine the effective address
```

Forcing Direct or Extended Addressing

The 6809 assembler has a mechanism for forcing the assembler to perform either direct or extended addressing. Under normal conditions, the assembler will use direct addressing when possible. To force the assembler to use extended addressing no matter what the conditions, simply precede the operand with a greater than sign ('>'). For example, suppose the DP register was set to $00 (this is the default on reset of the cpu), and that we have a label, BUFPNT, which is at memory location $0010. Normally the instruction:

```
LDX  BUFPNT
```

would be assembled with direct addressing. If we wished to force extended addressing we could simply enter:

```
LDX  >BUFPNT
```

and the assembler would use extended addressing.

The same capability exists for forcing direct addressing by preceding the operand with a less than sign ('<'). For example:

```
LDX  <BUFPNT
```

would force direct addressing. Note that in both cases the greater than or less than sign must be the first character in the operand.

The Assembler Instruction Set

This section contains a brief listing of all the mnemonics accepted by the 6809 assembler. They are listed in four sections, standard 6809 with alternate 6800, 6800 mnemonics not found in 6809, 6801 mnemonics, and non-standard convenience mnemonics. Before the listing, we must setup some notational conventions:

(P)	Operand containing immediate, extended, direct, or indexed addressing.
(Q)	Operand containing extended, direct, or indexed addressing.
(T)	Operand containing indexed addressing only.
R	Any register specification: A, B, X, Y, U, S, PC, CC, DP, or D.
dd	8 bit data value
dddd	16 bit data value

6809 Mnemonics with 6800 Alternates

ABX — Add B into X
SOURCE FORM: ABX

ADC — Add with carry into register
SOURCE FORM: ADCA (P); ADCB (P)
6800 ALTERNATES: ADC A (P); ADC B (P)

ADD — Add into register
SOURCE FORM: ADDA (P); ADDB (P); ADDD (P)
6800 ALTERNATES: ADD A (P); ADD B (P)

AND — Logical 'AND' into register
SOURCE FORM: ANDA (P); ANDB (P)
6800 ALTERNATES: AND A (P); AND B (P)

ANDCC — Logical 'AND' immediate into CC
SOURCE FORM: ANDCC #dd

ASL — Arithmetic shift left
SOURCE FORM: ASLA; ASLB; ASL (Q)
6800 ALTERNATES: ASL A; ASL B

ASR — *Arithmetic shift right*
SOURCE FORM: ASRA; ASRB; ASR (Q)
6800 ALTERNATES: ASR A; ASR B

BCC, LBCC — Branch (short or long) if carry clear
SOURCE FORM: BCC dd; LBCC dddd

BCS, LBCS — Branch (short or long) if carry set
SOURCE FORM: BCS dd; LBCS dddd

BEQ, LBEQ — Branch (short or long) if equal
SOURCE FORM: BEQ dd; LBEQ dddd

BGE, LBGE — Branch (short or long) if greater than or equal
SOURCE FORM: BGE dd; LBGE dddd

BGT, LBGT — Branch (short or long) if greater than
SOURCE FORM: BGT dd; LBGT dddd

BHI, LBHI — Branch (short or long) if higher
SOURCE FORM: BHI dd; LBHI dddd

BHS, LBHS — Branch (short or long) if higher or same
SOURCE FORM: BHS dd; LBHS dddd

BIT — Bit test
SOURCE FORM: BITA (P); BITB (P)
6800 ALTERNATES: BIT A (P); BIT B (P)

BLE, LBLE — Branch (short or long) if less than or equal to
SOURCE FORM: BLE dd; LBLE dddd

BLO, LBLO — Branch (short or long) if lower
SOURCE FORM: BLO dd; LBLO dddd

BLS, LBLS — Branch (short or long) if lower or same
SOURCE FORM: BLS dd; LBLS dddd

BLT, LBLT — Branch (short or long) if less than
SOURCE FORM: BLT dd; LBLT dddd

BMI, LBMI — Branch (short or long) if minus
SOURCE FORM: BMI dd; LBMI dddd

BNE, LBNE — Branch (short or long) if not equal
SOURCE FORM: BNE dd; LBNE dddd

BPL, LBPL — Branch (short or long) if plus
SOURCE FORM: BPL dd; LBPL dddd

BRA, LBRA Branch (short or long) always
SOURCE FORM: BRA dd; LBRA dddd

BRN, LBRN Branch (short or long) never
SOURCE FORM: BRN dd; LBRN dddd

BSR, LBSR Branch (short or long) to subroutine
SOURCE FORM: BSR dd; LBSR dddd

BVC, LBVC Branch (short or long) if overflow clear
SOURCE FORM: BVC dd; LBVC dddd

BVS, LBVS Branch (short or long) if overflow set
SOURCE FORM: BVS dd; LBVS dddd

CLR Clear
SOURCE FORM: CLRA; CLRB; CLR (Q)
6800 ALTERNATES: CLR A; CLR B

CMP Compare
SOURCE FORM: CMPA (P); CMPB (P); CMPD (P); CMPX (P);
CMPY (P); CMPU (P); CMPS (P)
6800 ALTERNATES: CMP A (P); CMP B (P); CPX (P)

COM Complement (One's complement)
SOURCE FORM: COMA; COMB; COM (Q)
6800 ALTERNATES: COM A; COM B

CWAI Clear and wait for interrupt
SOURCE FORM: CWAI #dd

DAA Decimal adjust accumulator A
SOURCE FORM: DAA

DEC Decrement
SOURCE FORM: DECA, DECB, DEC (Q)
6800 ALTERNATES: DEC A; DEC B

EOR Exclusive 'OR'
SOURCE FORM: EORA (P); EORB (P)
6800 ALTERNATES: EOR A (P); EOR B (P)

EXG Exchange registers
SOURCE FORM: EXG R1,R2

INC Increment
SOURCE FORM: INCA, INCB, INC (Q)
6800 ALTERNATES: INC A; INC B

JMP — Jump to address
SOURCE FORM: JMP dddd

JSR — Jump to subroutine at address
SOURCE FORM: JSR dddd

LD — Load register from memory
SOURCE FORM: LDA (P); LDB (P); LDD (P); LDX (P);
LDY (P); LDU (P); LDS (P)
6800 ALTERNATES: LDAA (P); LDAB (P); LDA A (P); LDA B (P)

LEA — Load effective address
SOURCE FORM: LEAX (T); LEAY (T); LEAU (T); LEAS (T)

LSL — Logical shift left
SOURCE FORM: LSLA; LSLB; LSL (Q)

LSR — Logical shift right
SOURCE FORM: LSRA; LSRB; LSR (Q)
6800 ALTERNATES: LSR A; LSR B

MUL — Multiply accumulators
SOURCE FORM: MUL

NEG — Negate (Two's complement)
SOURCE FORM: NEGA; NEGB; NEG (Q)
6800 ALTERNATES: NEG A; NEG B

NOP — No operation
SOURCE FORM: NOP

OR — Inclusive 'OR' into register
SOURCE FORM: ORA (P); ORB (P)
6800 ALTERNATES: ORAA (P); ORAB (P); ORA A (P); ORA B (P)

ORCC — Inclusive 'OR' immediate into CC
SOURCE FORM: ORCC #dd

PSHS — Push registers onto system stack
SOURCE FORM: PSHS (register list); PSHS #dd
6800 ALTERNATES: PSHA; PSHB; PSH A; PSH B

PSHU — Push registers onto user stack
SOURCE FORM: PSHU (register list); PSHU #dd

PULS — Pull registers from system stack
SOURCE FORM: PULS (register list); PULS #dd
6800 ALTERNATES: PULA; PULB; PUL A; PUL B

PULU — Pull registers from user stack
SOURCE FORM: PULU (register list); PULU #dd

ROL — Rotate left
SOURCE FORM: ROLA; ROLB; ROL (Q)
6800 ALTERNATES: ROL A; ROL B

ROR — Rotate right
SOURCE FORM: RORA; RORB; ROR (Q)
6800 ALTERNATES: ROR A; ROR B

RTI — Return from interrupt
SOURCE FORM: RTI

RTS — Return from subroutine
SOURCE FORM: RTS

SBC — Subtract with borrow
SOURCE FORM: SBCA (P); SBCB (P);
6800 ALTERNATES: SBC A (P); SBC B (P)

SEX — Sign extend
SOURCE FORM: SEX

ST — Store register into memory
SOURCE FORM: STA (P); STB (P); STD (P); STX (P);
STY (P); STU (P); STS (P)
6800 ALTERNATES: STAA (P); STAB (P); STA A (P); STA B (P)

SUB — Subtract from register
SOURCE FORM: SUBA (P); SUBB (P); SUBD (P)
6800 ALTERNATES: SUB A (P); SUB B (P)

SWI — Software interrupt
SOURCE FORM: SWI

SWI2 — Software interrupt 2
SOURCE FORM: SWI2

SWI3 — Software interrupt 3
SOURCE FORM: SWI3

SYNC — Synchronize to interrupt
SOURCE FORM: SYNC

TFR — Transfer register to register
SOURCE FORM: TFR R1,R2

TST — Test
SOURCE FORM: TSTA; TSTB; TST (Q)
6800 ALTERNATES: TST A; TST B

Simulated 6800 Instructions

ABA	Add B to A
CBA	Compare B to A
CLC	Clear carry bit
CLI	Clear interrupt mask
CLV	Clear overflow bit
DES	Decrement stack pointer
DEX	Decrement X
INS	Increment stack pointer
INX	Increment X
SBA	Subtract B from A
SEC	Set carry bit
SEI	Set interrupt mask
SEV	Set overflow bit
TAB	Transfer A to B
TAP	Transfer A to CC
TBA	Transfer B to A
TPA	Transfer CC to A
TSX	Transfer S to X
TXS	Transfer X to S
WAI	Wait for interrupt

Simulated 6801 Mnemonics

ASLD	Arithmetic shift left D
LSRD	Logical shift right D
PSHX	Push the X register
PULX	Pull the X register
LDAD	Load accumulator D from memory
STAD	Store accumulator D into memory

Convenience mnemonics

BEC,LBEC	Branch (short or long) if error clear
BES,LBES	Branch (short or long) if error set
CLF	Clear FIRQ interrupt mask
CLZ	Clear zero condition code bit
SEF	Set FIRQ interrupt mask
SEZ	Set zero condition code bit

V. STANDARD DIRECTIVES OR PSEUDO-OPS

Besides the standard machine language mnemonics, the TSC assembler supports several directives or pseudo-ops. These are instructions for the assembler to perform certain operations, and are not directly assembled into code. There are four types of directives in this assembler, those associated with conditional assembly, those associated with macros, those associated with segmented binary files, and those which generally can be used anywhere which we shall call "standard directives". This section is devoted to descriptions of these standard directives which are briefly listed here:

org	setdp
end	pag
rmb	spc
fcb	nam or ttl
fdb	sttl
fcc	err
equ	rpt
set	lib
reg	opt
rzb	sys
info	setid

Descriptions of each directive and its use follow later.

Though the other types of directives are explained in other sections, for the sake of completeness we shall list them here:

Conditional Directives	Macro Directives	Segmentation Directives
if	macro	text
ifn	endm	data
ifc	exitm	bss
ifnc	dup	base
else	endd	
endif		

org

The 'org' statement is used to set a new code 'Origin'. This simply means that a new address is set into the location counter (or program counter) so that subsequent code will be placed at the new location. The form is as follows:

```
org  <expression>
```

No label may be placed on an 'org' statement and no code is produced. If no 'org' statement appears in the source, an origin of 0000 is assumed. If the object code file is to be segmented as in the section on Object Code Production, 'org' takes on added meaning and the user should refer to that section for details.

end

The 'end' directive is used to signify the end of the input source program. In this assembler the 'end' statement is actually ignored except for a means of supplying a transfer address as explained shortly. The assembler terminates the assembly when the end of file condition is detected on the input file(s). No label is allowed and no code is generated. An expression may be given (as shown below) as the transfer address to be placed in a binary file. It is optional, and if supplied when no binary file is being produced, will be ignored. If more than one 'end' statements with transfer addresses are encountered in a file, all but the last transfer address will be ignored.

```
end   [<expression>]
```

Note that an 'end' statement is not strictly required, but is the only means of getting a transfer address appended to a binary output file.

rmb

The 'rmb' or Reserve Memory Bytes directive is used to reserve areas of memory for data storage. The number of bytes specified by the expression in the operand are skipped during assembly. No code is produced in those memory location and therefore the contents are undefined at run time. The proper useage is shown here:

```
[<label>]  rmb  <expression>
```

The label is optional, and the expression is a 16 bit quantity.

rzb

The 'rzb' or Reserve Zeroed Bytes directive is used to initialize an area of memory with zeroes. Beginning with the current PC location, the number of bytes specified will be set to zero. The proper syntax is:

```
[<label>]  rzb  <expression>
```

where the expression can be any value from 1 to 65,535. This directive does produce object code.

fcb

The 'fcb' or Form Constant Byte directive is used to set associated memory bytes to some value as determined by the operand. 'fcb' may be used to set any number of bytes as shown below:

```
[<label>]  fcb  <expr. 1>,<expr. 2>,...,<expr. n>
```

Where <expr. x> stands for some expression. Each expression given (separated by commas) is evaluated to 8 bits and the resulting quantities are stored in successive memory locations. The label is optional.

fdb

The 'fdb' or Form Double Byte directive is used to setup 16 bit quantites in memory. It is exactly like the 'fcb' directive except that 16 bit quantities are evaluated and stored in memory for each expression given. The form of the statement is:

```
[<label>]  fdb  <expr. 1>,<expr. 2>,...,<expr. n>
```

Again, the label field is optional.

fcc

The 'fcc' or Form Constant Character directive allows the programmer to specify a string of ASCII characters delimited by some non-alphanumeric character such as a single quote. All the characters in the string will be converted to their respective ASCII values and stored in memory, one byte per character. Some valid examples follow:

```
label1  fcc  'This is an fcc string'
label2  fcc  .so is this.
        fcc  /Labels are not required./
```

There is another method of using 'fcc' which is a deviation from the standard Motorola definition of this directive. This allows you to place certain expressions on the same line as the standard 'fcc' delimited string. The items are separated by commas and are evaluated to 8 bit results. In some respects this is like the 'fcb' directive. The difference is that in the 'fcc' directive, expressions must begin with a letter, number or dollar-sign whereas in the 'fcb' directive any valid expression will work. For example, %10101111 would be a valid expression for an 'fcb' but not for an 'fcc' since the percent-sign would look like a delimiter and the assembler would attempt to produce 8 bytes of data from the 8 ASCII characters which follow (an 'fcc' string). The dollar-sign is an exception to allow hex values such as $0D (carriage return) to be inserted along with strings. Some examples follow:

```
intro  fcc  'This string has CR & LF',$D,$A
       fcc  'string 1',0,'string 2'
       fcc  $04,label,/delimited string/
```

Note that more than one delimited string may be placed on a line as in the second example.

equ

The 'equ' or Equate directive is used to equate a symbol to the expression given in the operand. No code is generated by this statement. Once a symbol has been equated to some value, it may not be changed at a later time in the assembly. The form of an equate statement is as follows:

```
<label>  equ  <expression>
```

The label is strictly required in equate statements.

set

The 'set' directive is used to set a symbol to the value of some expression, much as an 'equ' directive. The difference is that a symbol may be 'set' several times within the source (to different values) while a symbol may be Equated only once. If a symbol is 'set' to several values within the source, the current value of the symbol will be the value last 'set'. The statement form is:

```
<label>  set  <expression>
```

The label is strictly required and no code is generated.

reg

The 'reg' directive allows the user to setup a list of registers for use by the push and pull instructions. This list is represented by a value and the value is equated to the label supplied. In this respect, the 'reg' directive is similar to the 'equ' directive. The correct form of the 'reg' directive is:

```
<label>  reg  <register list>
```

As an example, suppose a program has a large number of occurences of the following instructions:

```
pshs  a,b,y,u,dp
puls  a,b,y,u,dp
```

To make things more convenient and less error prone the 'reg' directive could be used as shown here:

```
rlist2   reg  a,b,y,u,dp
```

Now all the pushes and pulls referred to above could be accomplished with the statements:

```
pshs #rlist2
puls #rlist2
```

Of course, the register list may still be typed out on push and pull instructions or an immediate value (with the desired bit pattern) may be specified.

setdp

The 'setdp' or Set Direct Page directive allows the user to set which memory page the assembler will use for the direct page addressing mode. The correct format is as follows:

```
setdp  [<page value>]
```

As an example, if "setdp $D0" is encountered, the assembler will then use direct addressing for any address in the range of $D000 to $D0FF. It is important to note that this directive does not actually affect the contents of the direct page register. The value set is what will be used at assembly time to determine direct addressing, but it is up to the user to be sure the DP register corresponds at run time. If there is no <page value> supplied, direct addressing will be disabled and all addresses will be full 16 bit values. Any number of 'setdp' instructions may occur in a program. The default value is page 0 (for 6800 compatibility).

pag

The 'pag' directive causes a page eject in the output listing and prints a header at the top of the new page. Note that the 'pag' option must be enabled in order for this directive to take effect. It is possible to assign a new number to the new page by specifying such in the operand field. If no page number is specified, the next consecutive number will be used. No label is allowed and no code is produced. The 'pag' operator itself will not appear in the listing unless some sort of error is encountered. The proper form is:

```
pag  [<expression>]
```

Where the expression is optional. The first page of a listing does not have the header printed on it and is considered to be page 0. The intention here is that all options, title, and subtitle may be setup and followed by a 'pag' directive to start the assembled listing at the top of page 1 without the option, title, or subtitle instructions being in the way.

spc

The 'spc' or Space directive causes the specified number of spaces (line feeds) to be inserted into the output listing. The general form is:

```
spc  [<space count>[,<keep count>]]
```

The space count can be any number from 0 to 255. If the page option is selected, 'spc' will not cause spacing past the top of a new page. The <keep count> is optional and is the number of lines which the user wishes to keep together on a page. If there are not enough lines left on the current page, a page eject is performed. If there are <keep count> lines left on the page (after printing <space count> spaces), output will continue on the current page. If the page option is not selected, the <keep count> will be ignored. If no operand is given (ie. just the directive 'spc'), the assembler will default to one blank line in the output listing.

ttl or nam

The 'ttl' or 'nam' directive allows the user to specify a title or name to the program being assembled. This title is then printed in the header at the top of each output listing page if the page option is selected. If the page option is not selected, this directive is ignored. The proper form is:

```
ttl  <text for the title>
        or
nam  <text for the title>
```

All the text following the 'ttl' or 'nam' directive (excluding leading spaces) is placed in the title buffer. Up to 32 characters are allowed with any excess being ignored. It is possible to have any number of

'ttl' or 'nam' directives in a source program. The latest one encountered will always be the one used for printing at the top of the following page(s).

sttl

The 'sttl' or Subtitle directive is used to specify a subtitle to be printed just below the header at the top of an output listing page. It is specified much as the 'ttl' directive:

```
sttl  <text for the subtitle>
```

The subtitle may be up to 52 characters in length. If the page option is not selected, this directive will be ignored. As with the 'ttl' option, any number of 'sttl' directives may appear in a source program. The subtitle can be disabled or turned off by an 'sttl' command with no text following.

err

The 'err' directive may be used to insert user-defined error messages in the output listing. The error count is also bumped by one. The proper form is:

```
err  <message to be printed>
```

All text past the 'err' directive (excluding leading spaces) is printed as an error message (it will be preceded by three asterisks) in the output listing. Note that the 'err' directive line itself is not printed. A common use for the 'err' directive is in conjunction with conditional assembly such that some user-defined illegal condition may be reported as an error.

info

The 'info' directive allows the user to store textual comments in a binary file. This is a UniFLEX supported feature in that under UniFLEX a user can execute the command 'info' and view the text at the terminal. The assembler's 'info' directive places all text following the 'info' command (excluding leading spaces) into a temporary file called '/tmp/asmbinfoxxxxx', where xxxxx represents the current task number. At the end of the assembly, all text stored in this temporary file is appropriately copied into the normal binary file and the temporary file is then deleted. Syntax is as follows:

```
info  This is a comment for the binary file.
info  It is a convenient way of inserting version nos.
info  Version X.XX - Released XX/XX/XX
```

Any number of 'info' directives may be inserted at any point in the source listing. No label is allowed and no actual binary code is produced.

rpt

The 'rpt' or Repeat directive causes the succeeding line of source to be repeated some specified number of times. The syntax is as follows:

```
rpt  <count>
```

where <count> may be any number from 1 to 127. For example, the following two lines:

```
rpt  4
aslb
```

would produce an assembled output of:

```
aslb
aslb
aslb
aslb
```

Some directives, such as 'if' or 'macro', may not be repeated with the 'rpt' command. These cases are where it is illogical or impractical to do so. If attempted, the 'rpt' will simply be ignored.

lib

The 'lib' or Library directive allows the user to specify an external file for inclusion in the assembled source output. Under normal conditions, the assembler reads all input from the file(s) specified on the calling line. The 'lib' directive allows the user to temporarily obtain the source lines from some other file. When all the lines in that external file have been read and assembled, the assembler resumes reading of the original source file. The proper syntax is:

```
lib  <file spec>
```

where <file spec> is a standard UniFLEX file specification. The assembler first looks for the specified file in the current directory. If not found it then looks for a directory named "lib" in the current directory. If found, the assembler attempts to find the specified file in that "lib" directory. If not found there, the assembler makes a third and final attempt to find the specified file by looking in the directory "/lib". If not found in any of these three directories, the assembler gives up and reports an error. Any 'end' statements found in the file called by the 'lib' directive are ignored. The 'lib' directive

line itself does not appear in the output listing. Any number of 'lib' instructions may appear in a source listing. It is also possible to nest 'lib' files up to 12 levels. Nesting refers to the process of placing a 'lib' directive within the source that is called up by another 'lib' directive. In other words, one 'lib' file may call another.

setid

The 'setid' directive allows the programmer to turn on the "setid" bit in the permissions field of the binary file being produced. By default this bit is turned off. When set, this bit temporarily gives any user of the command the same permissions which the owner has. For more information on the function of the "setid" bit, consult your UniFLEX documentation. The syntax is quite simply:

```
setid
```

There is no operand required.

sys

The 'sys' or system call directive allows the programmer to setup a system call to UniFLEX. Such a call is comprised of an SWI3 instruction followed by a one byte function code optionally followed by 16-bit parameter values. This directive automatically inserts the SWI3 and then obtains the function code and any other parameters from the operand field. Proper usage is shown here:

```
sys  <function>,<parameter1>,<parameter2>,...
```

The <function> and <parameter> values may be any legal expression. <function> will be stored as 8-bits while all <parameters> will be stored as 16-bits.

opt

The 'opt' or Option directive allows the user to choose from several different assembly options which are available to him. These options are generally related to the format of the output listing and object code. The options which may be set with this command are listed below. The proper form of this instruction is:

```
opt  <option 1>,<option 2>,...,<option n>
```

Note that any number of options may be given on one line if separated by commas. No label is allowed and no spaces or tabs may be imbedded in the option list. The options are set during pass two. If contradicting options are specified, the last one appearing takes precedence. If a particular option is not specified, the default case for that option takes effect. The default cases are signified below by an asterisk.

The allowable options are:

```
pag*  enable page formatting and numbering
nop   disable pagination

con   print conditionally skipped code
noc*  suppress conditional code printing

mac*  print macro calling lines
nom   suppress printing of macro calls

exp   print macro expansion lines
noe*  suppress macro expansion printing

lis*  print an assembled listing
nol   suppress output of assembled listing
```

* denotes default option and is not part of option name

The 'lis' and 'nol' options may be used to selectively turn parts of a program listing on or off as desired. If the '+l' command line option is specified, however, the 'lis' and 'nol' options are overridden and no listing occurs.

VI. CONDITIONAL ASSEMBLY

This assembler supports "conditional assembly" or the ability to assemble only certain portions of your source program depending on the conditions at assembly time. Conditional assembly is particularly useful in situations where you might need several versions of a program with only slight changes between versions.

As an example, suppose we required a different version of some program for 4 different systems whose output routines varied. Rather than prepare four different source listings, we could prepare one which would assemble a different set of output routines depending on some variable which was set with an 'equ' directive near the beginning of the source. Then it would only be necessary to change that one 'equ' statement to produce any of the four final programs. This would make the software easier to maintain, as besides only needing to keep track of one copy of the source, if a change is required in the body of the program, only one edit is required to update all versions.

The 'if-endif' Clause

In its simplest form, conditional assembly is performed with two directives: 'if' and 'endif'. The two directives are placed in the source listing in the above order with any number of lines of source between. When the assembler comes across the 'if' statement, it evaluates the expression associated with it (we will discuss this expression in a moment) and if the result is true, assembles all the lines between the 'if' and 'endif' and then continues assembling the lines after the 'endif'. If the result of the expression is false, the assembler will skip all lines between the 'if' and 'endif' and resume assembly of the lines after the 'endif'. The proper syntax of these directives is as follows:

```
if  <expression>
    .
    .   conditional code goes here
    .
endif
```

The 'endif' requires no additional information but the 'if' requires an expression. This expression is considered FALSE if the 16-bit result is equal to zero. If not equal to zero, the expression is considered TRUE.

A more powerful conditional assembly construct is possible with the 'else' directive. The 'else' directive may be placed between the 'if' and 'endif' statements. Its effect is to switch the sense of the test for the lines between it and the 'endif'. In effect, the lines of source between the 'if' and 'endif' are split into two groups by the 'else' statement. Those lines before the 'else' are assembled if the expression is true while those after (up to the 'endif') are ignored. If the expression is false, the lines before the 'else' are ignored while those after it are assembled. The 'if-else-endif' construct

appears as follows:

```
        if  <expression>
          .
          .   this code assembled if expression is true
          .
        else
          .
          .   this code assembled if expression is false
          .
        endif
```

The 'else' statement does not require an operand and there may be only one 'else' between an 'if-endif' pair.

It is possible to nest 'if-endif' clauses (including 'else's). That is to say, an 'if-endif' clause may be part of the lines of source found inside another 'if-endif' clause. You must be careful, however, to terminate the inner clause before the outer.

There is another form of the conditional directive, namely 'ifn' which stands for "if not". This directive functions just like 'if' except that the sense of the test is reversed. Thus the code immediately following is assembled if the result of the expression is NOT TRUE. An 'ifn-else-endif' clause appears as follows:

```
        ifn  <expression>
          .
          .   this code assembled if expression is FALSE
          .
        else
          .
          .   this code assembled if expression is TRUE
          .
        endif
```

The 'ifc-endif' Clause

Another form of conditional assembly is very similar to the 'if-endif' clause defined above, but depends on a comparison of two strings for its conditionality instead of a true-false expression. This type of conditional assembly is done with the 'ifc' and 'ifnc' directives (for "if compare" and "if not compare") as well as the 'else' and 'endif' discussed above. Thus for the 'ifc' directive we have a clause like:

```
        ifc  <string 1>,<string 2>
          .
          .   this code assembled if strings are equal
          .
        else
          .
          .   this code assembled if strings are not equal
          .
        endif
```

As can be seen, the two strings are separated by a comma. There are two types of strings, one enclosed by delimiters the other not. The delimited type may use either a single quote (') or double quote (") as the delimiter. This type of string is made up of all the characters after the first delimiter until the second delimiter is found. The second type of string is simply a group of characters, starting with a non-space and containing no spaces or commas. Thus if you need spaces or commas in a string, you must use the delimited type of string. It is possible to specify a null string by placing two delimiters in a row or by simply leaving the string out completely. Note that there may be no spaces after string 1 and before the separating comma nor after the comma and before string 2. As with 'ifn', the 'ifnc' directive simply reverses the sense of the test such that code immediately following an 'ifnc' directive would be assembled if the strings did NOT compare.

A common application of this type of conditional assembly is in macros (defined in the next section) where one or both of the strings might be a parameter passed into the macro.

The 'if-skip' Clause

The 'if-skip' type of conditional assembly is a method which does not use (in fact does not allow) a related 'endif' or 'else'. Instead, the assembler is caused to skip a specified number of lines of source depending on the result of the expression or string comparison.

IMPORTANT NOTE:
This type of conditional assembly is
ONLY allowed within the body of a macro.
Any use of it outside a macro will
result in an error. Macros are defined
in the next section.

As before, the possible directives are: 'if', 'ifn', 'ifc', and 'ifnc'. This type of conditional assembly is performed with a single instruction. Instead of code being assembled on a true result, the specified number of lines are SKIPPED. This number of lines can be in a forward or reverse direction. The syntax is as shown:

```
if   <expression>,<skip count>
                or
ifc  <string 1>,<string 2>,<skip count>
```

The skip count must be a decimal number between 1 and 255. It may be preceded by a plus or minus sign. A positive number produces a forward skip while a negative number produces a backwards skip. A skip count of zero has no effect (the instruction following the 'if' directive will be executed next). A skip count of one will cause the second line after the 'if' statement to be the next one executed (the one line directly following the 'if' statement is ignored). A skip count of negative one will cause the line just before the skip count to be the next one executed. The assembler will not skip past the end or beginning of the macro which contains the 'if-skip' statement. If a skip count is specified which is beyond these limits, the assembler will be left

pointing to the last statement in the macro or the first, depending on whether the skip count was positive or negative. There can be no spaces before or after the comma which separates the skip count from the expression or from string 2.

IMPORTANT NOTE

In order for conditionals to function properly, they must capable of evaluation in pass one so that the same result will occur in pass two. Thus if labels are used in a conditional expression, they must have been defined in the source before the conditional direcive is encountered.

VII. MACROS

A macro is a facility for substituting a whole set of instructions and parameters in place of a single instruction or call to the macro. There are always two steps to the use of macros, the definition and the call. In the definition we specify what set of instructions make up the body of the macro and assign a name to it. This macro may then be called by name with a single assembler instruction line. This single line is replaced by the body of the macro or the group of lines which were defined as the macro. This replacement of the calling line with the macro body is called the macro "expansion". It is also possible to provide a set of parameters with the call which will be substituted into the desired areas of the macro body.

A simple example will assist in the understanding of macros. Let us define a macro which will shift the 'D' register left four places. This is such a simple operation that it does not really require or make effective use of macros, but it will suffice for learning purposes. In actuality this routine would probably be written in-line or, if required often, written as a subroutine.

The first step is to define the macro. This must be done BEFORE THE FIRST CALL to the macro. It is good practice to define all macros early in a program. The definition is initiated with a 'macro' directive and terminated by an 'endm' directive. The definition of our example would be as follows:

```
asld4   macro
        aslb
        rola
        aslb
        rola
        aslb
        rola
        aslb
        rola
        endm
```

The first line is the 'macro' directive. Note that the name of the macro is specified with this directive by placing it in the label field. This macro name should follow all the rules for labels. It will NOT be placed in the symbol table, but rather in a macro name table. The body of the macro follows and is simply lines of standard assembly source which shift the 'D' register left four places. The definition is terminated by the 'endm' directive.

When this macro definition is encountered during pass 1, the assembler will not actually assemble the source, but instead copy it into a buffer for future access when the macro is called. During pass 2 this definition is ignored.

At this point we may continue with our assembly program and when we desire to have the 'D' register shifted left four places, simply call the macro as follows:

```
        .
        .
lda     value
ldb     value+1
asld4                   here is the macro call
std     result
        .
        .
```

You can see that calling a macro consists of simply placing its name in the mnemonic field of a source line. When the assembler sees the above call, it realizes that the instruction is not a standard 6809 mnemonic, but rather a macro that has been previously defined. The assembler will then replace the asld4 with the lines which make up the body of that macro or "expand" the macro. The result would be the following assembled code:

```
        .
        .
lda     value
ldb     value+1
aslb                    the body of the macro
rola                    replaces the call
aslb
rola
aslb
rola
aslb
rola
std     result
        .
        .
```

You should note that a macro call differs from a subroutine call in that the macro call results in lines of code being placed in-line in the program where a subroutine call results in the execution of the routine at run-time. Five calls to a subroutine still only requires one copy of the subroutine while five calls to a macro results in five copies of the macro body being inserted into the program.

Parameter Substitution

If macros were limited to what was described above, they would probably not be worth the space it took to implement them in the assembler. The real power of macros comes in "parameter substitution". By that we mean the ability to pass parameters into the macro body from the calling line. In this manner, each expansion of a macro can be different.

As an example, suppose we wanted to add three 16-bit numbers found in memory and store the result in another memory location. A simple macro to do this (we shall call it add3) would look like this:

```
add3  macro
      ldd    loc1    get first value in 'D'
      addd   loc2    add in second value
      addd   loc3    add in third value
      std    result  store result
      endm
```

Now let's assume we need to add three numbers like this in several places in the program, but the locations from which the numbers come and are to be stored are different. We need a method of passing these locations into the macro each time it is called and expanded. That is the function of parameter substitution. The assembler lets you place up to nine parameters on the calling line which can be substituted into the expanded macro. The proper form for this is:

```
macnam  <prm.1>,<prm.2>,<prm.3>,...,<prm.9>
```

where "macnam" is the name of the macro being called. Each parameter may be one of two types: a string of characters enclosed by like delimiters and a string of characters not enclosed by delimiters which contains no embedded spaces or commas. The delimiter for the first type may be either a single quote (') or a double quote (") but the starting and ending delimiter of a particular string must be the same. A comma is used to separate the parameters. These parameters are now passed into the macro expansion by substituting them for 2-character "dummy parameters" which have been placed in the macro body on definition. These 2-character dummy parameters are made up of an ampersand (&) followed by a single digit representing the number of the parameter on the calling line as seen above. Thus any occurence of the dummy parameter, "&1", would be replaced by the first parameter found on the calling line.

Let's re-do our add3 macro to demonstrate this process. The definition of add3 now looks like this:

```
add3  macro
      ldd    &1    get first value in 'D'
      addd   &2    add in second value
      addd   &3    add in third value
      std    &4    store result
      endm
```

Now to call the macro we might use a line like:

```
add3  loc1,loc2,loc3,result
```

When this macro was expanded, the &1 would be replaced with loc1, the &2 would be replaced with loc2, etc. The resulting assembled code would appear as follows:

```
ldd    loc1    get first value in 'D'
addd   loc2    add in second value
addd   loc3    add in third value
std    result  store result
```

Another call to the macro might be:

```
add3  ace,two,loc3,loc1
```

which would result in the following expansion:

```
ldd    ace     get first value in 'D'
addd   two     add in second value
addd   loc3    add in third value
std    loc1    store result
```

Now you should begin to see the power of macros.

There is actually a tenth parameter which may be passed into a macro represented by the dummy parameter "&0". It is presented on the calling line as so:

```
<prm.0>  macnam  <prm.1>,<prm.2>,<prm.3>,...,<prm.9>
```

This parameter is somewhat different from the others in that it must be a string of characters that conform to the rules for any other assembly language label since it is found in the label field. It is in fact a standard label which goes into the symbol table and can be used in other statement's operands like any other label.

Ignoring a Dummy Parameter

There may be times when a programmer wishes to have an ampersand followed by a number in a macro which is not a dummy parameter and should therefore not be replaced with a parameter string. An example would be an expression where the value of 'mask' was to be logically 'anded' with the number 4. The expression would appear like:

```
mask&4
```

If this expression were in a macro, upon expansion the &4 would be replaced with the fourth parameter on the calling line. It is possible to prevent this, however, by preceding the ampersand with a backslash (\) like this:

```
mask\&4
```

When the assembler expands the macro containing this expression, it will recognize the backslash, remove it, and leave the ampersand and following number intact.

Another case where this can be useful is when a macro is defined within a macro (that is possible!) and you wish to place dummy parameters in the inner macro.

The EXITM Directive

Sometimes it is desireable to exit a macro prematurely. The 'exitm' directive permits just that. During expansion of a macro, when an 'exitm' command is encountered the assembler immediately skips to the 'endm' statement and terminates the expansion. This probably does not seem logical, and is not except when used with conditional assembly.

To portray the use of 'exitm', assume we have some macro called 'xyz' which has two parts. The first part should always be expanded, but the second should only be expanded in certain cases. We could use 'exitm' and the 'ifnc' directives to accomplish this as follows:

```
xyz  macro
       .
       .       code that should always be generated
       .
     ifnc &2,yes
     exitm
     endif
       .
       .      code that is only sometimes generated
       .
     endm
```

The following calls would result in the second part being expanded or producing code:

```
xyz  "parameter 1",yes
xyz  "parameter 1","yes"
xyz  0,yes
```

while all of the following would result in the second part not being expanded:

```
xyz  "parameter 1",no
xyz  junk,no
xyz  junk
xyz  prm1,prm2
```

The 'exitm' directive itself requires no operand.

The DUP and ENDD Directives

There is another type of assembler construct which may only be used inside of a macro, called the 'dup-endd' clause. The assembler will duplicate the lines placed between the 'dup' and 'endd' (end dup) instructions some specified number of times. The proper form is:

```
        dup  <dup count>
         .
         .          code to be duplicated
         .
        endd
```

where the <dup count> is the number of times the code should be duplicated. The <dup count> may be any valid expression, but must be in the range of 1 to 255 decimal. Note that 'dup-endd' clauses may NOT be nested. That is to say, one 'dup-endd' clause may not be placed inside another.

As an example, let's take our first example in this section on macros and spruce it up a little. Assume we want a macro that will shift the 'D' register to the left 'x' places where 'x' can vary in different calls to the macro. The 'dup-endd' construct will work nicely for this purpose as seen here:

```
    asldx  macro
           dup    &1
           aslb
           rola
           endd
           endm
```

Now to shift the 'D' register left four places we call the macro with:

```
           asldx   4
```

To shift it left 12 places we simply use the instruction:

```
           asldx   12
```

And so on.

More on MACROS

A few more hints on using macros may be of value.

One important thing to remember is that parameter substitution is merely replacing one string (the dummy parameter or &x) with another (the parameter string on the calling line). You are not passing a value into a macro when you have a parameter of "1000", but rather a string of four characters. When the expanded source code of the macro is assembled, the string may be considered a value, but in the phase of parameter substitution it is merely a string of characters. An example macro will help clarify this point.

```
test  macro
      lda    #$&1     comment field is here
      ldb    l&1
      nop             parameters can even be
      nop             substituted into
      nop             comments &2
      &3              or they can be a mnemonic
&4    tst    m&1m     or label or inside a string
      endm
```

Now if this macro were called with the following command:

```
      test  1000,'like this',sex,"label"
```

The expanded source code would look like this:

```
      lda    #$1000   comment field is here
      ldb    l1000
      nop             parameters can even be
      nop             substituted into
      nop             comments like this
      sex             or they can be a mnemonic
label tst    m1000m   or label or inside a string
```

Note that in the 'lda' instruction the parameter "1000" is used as a number but in the 'ldb' and 'tst' instructions it is part of a label. The second parameter is not even substituted into the actual program, but rather into the comment field. The use of parameter number one in the 'tst' instructions shows that the dummy parameter does not have to be a stand alone item but can be anywhere in the line.

Another convenient method of using macros is in conjunction with the 'if-skip' type of conditional directive. With a negative or backward skip we can cause a macro to loop back on itself during expansion. A good example of this would be a case where the programmer must initialize one hundred consecutive memory locations to the numbers one through one hundred. This would be a very tedious task if all these numbers had to be setup by 'fcb' directives. Instead we can use a single 'fcb' directive, the 'if-skip' type of directive, and the 'set' directive to accomplish this task.

```
init   macro
count  set   0        initialize counter
count  set   count+1  bump by one
       fcb   count    set the memory byte
       if    count<100,-2
       endm
```

If you try this macro out, you will see that it expands into quite a bit of source if the macro expansions are being listed because the 3rd, 4th and 5th line are expanded for each of the one hundred times through. However, only one hundred bytes of object code are actually produced since lines 3 and 5 don't produce code.

This assembler does not allow local variables. If a label is specified on a line in the macro, you will receive a multiply defined symbol error if the macro is called more than once. There is a way to get around this shortcoming that is somewhat crude, but effective. That is to use a dummy parameter as a label and require the programmer to supply a different label name as that parameter each time the macro is called. For example, consider the following example:

```
shift macro
      pshs   d
      lda    &1
      ldb    #&2
&3    rola
      decb
      bne    &3
      sta    &1
      puls   d
      endm
```

Now if this macro was called with the line:

```
      shift  flag,3,call1
```

The resulting macro expansion would look like:

```
      pshs   d
      lda    flag
      ldb    #3
call1 rola
      decb
      bne    call1
      sta    flag
      puls   d
```

There is no problem here, but if the macro were called again with the same string for parameter 3 ('call1'), a multiply defined symbol error would result. Thus any subsequent calls to 'shift' must have a unique string for parameter 3 such as 'call2', 'call3', etc.

Important Notes on MACROS

1) A macro must be defined before the first call to it.

2) Macros can be nested both in calls and in definitions. That is, one macro may call another and one macro may be defined inside another.

3) Comment lines are stripped out of macros when defined to save storage space in the macro text buffer.

4) Local labels are not supported.

5) A macro cannot call a library file. That is, a 'lib' directive cannot appear within a macro.

6) No counting of parameters is done to be sure enough parameters are supplied on the calling line to satisfy all dummy parameters in the defined macro. If the body of a macro contains a dummy parameter for which no parameter is supplied on the calling line, the dummy parameter will be replaced with a null string or effectively removed.

7) The macro name table is searched before the mnemonic table. This means that a standard mnemonic or directive can be effectively redefined by replacing it with a macro of the same name.

8) Once a macro has been defined, it cannot be purged nor re-defined.

VIII. SPECIAL FEATURES

This section covers a few special features of the UniFLEX 6809 Assembler that don't seem to fit under any other specific category.

End of Assembly Information

Upon termination of an assembly, and before the symbol table is output, three items of information may be printed: the total number of errors encountered, the total number of warnings encountered, and the address of the last byte of code assembled.

The number of errors is always printed in the following manner:

```
0 Error(s) detected
```

The number of warnings are printed only if warnings have not been suppressed and if the number is greater than zero (ie. only if there was a warning):

```
0 Error(s) detected    2 Warning(s) reported
```

The last assembled address is printed only if the assembled output listing is turned off. It is actually the last address which the assembler's program counter register was pointing to, so there may not really be an assembled byte of code at this address. For example if the last instruction in a program except for the 'end' was an 'rmb' directive, the address would be the last byte reserved by that command. This information is presented as follows:

```
Last assembled address: 1055
```

The address is printed as a 4-digit hexadecimal value.

Absolute Address Printer

Specifying the '+a' option on the assembler calling line causes an absolute address to be printed in the object code field of the assembled output listing for all relative branch instructions. This is very convenient for debugging purposes, especially when long branches are used.

Excessive Branch or Jump Indicator

A mechanism has been included in the assembler to inform the programmer when a long branch or jump could be replaced by a short branch. The purpose is to allow size and speed optimization of the final code. This indicator is a greater-than sign placed just before the address of the long branch or jump instruction which could be shortened. The following section of code shows just how it looks:

```
 3420 B6   E004      inch    lda     $E004
 3423 84   01                anda    #$01
>3425 1027 FFF7              lbeq    inch
 3429 B6   E005              lda     $E005
 342C 81   20                cmpa    #$20
 342E 24   06                bhs     outch
>3430 BD   3436              jsr     outch
>3433 7E   3420              jmp     inch
 3436 34   04      outch     pshs    b
                               .
                               .
                               .
```

These indicator flags may be suppressed by turning off warnings.

Auto Fielding

The assembler automatically places the four fields of a source line (label, mnemonic, operand, and comment) in columns in the output assembled listing. This allows the programmer to edit a condensed source file without impairing the readability of the assembled listing. The common method of doing this is to separate the fields by only one space when editing. The assembled output places all mnemonics beginning in column 8, all opcodes beginning in column 15, and all comments beginning in column 25 assuming the previous field does not extend into the current one. There are a few cases where this automatic fielding can break down such as lines with errors, but these cases are rare and generally cause no problem.

Command Line Parameters

In the section on using the assembler at the beginning of this manual, command line parameters are discussed. That section describes how to place a parameter on the command line for passing into the source, but it does not elaborate on how the programmer tells the assembler where in the source to substitute those parameters. If you have read the section on macros, you should already understand the concept of parameter substitution. If not, read that section before continuing here.

Much as in macro parameter substitution, there are 2-character symbols or dummy parameters which, if placed anywhere in the source, will be replaced by the parameters supplied on the command line. In macros, the parameters from the macro calling line were substituted into the macro body during expansion of the macro. Here, the parameters from the command line are substituted into the source as it is read in from the disk. In macros, there were 10 possible parameters. Here there are three possible. The 2-character dummy parameters for these three are '&a', '&b', and '&c'. These correspond to the three possible command line parameters represented here:

```
++ asmb filename +options +a=prm.A +b=prm.B +c=prm.C
```

Here 'prm.A' represents parameter 'a' for substitution in place of '&a', 'prm.B' represents parameter 'b', etc.

Just as in macros, the dummy parameters can be ignored by placing a backslash directly in front of the ampersand. Thus the following line of edited source:

```
value  equ  mask\&count
```

would be read into the assembler as:

```
value  equ  mask&count
```

A quick example should help clarify the preceding descriptions. The following program contains one dummy parameter, '&a'.

```
* Routine to output to one of two ports
      opt   con
      ttl   Output Routine For Port #&a
      pag
      ifn   (&a=0)|(&a=1)
      err   Not a valid port number.
      else
      if    &a=0
acia  equ   $E000
      else
acia  equ   $E004
      endif
      endif
```

```
outch ldb   acia
      andb  #$02
      beq   outch
      sta   acia
      rts

      end
```

Now if this file were assembled with a command line like:

```
++ asmb file +bds +a=1
```

(assuming the file is very creatively called, 'file'), we would see the following assembled output:

```
Output Routine For Port #1              UniFLEX Assembler  Page   1


                                  ifn     (1=0)|(1=1)
                                  err     Not a valid port number.
                                  else
                                  if      1=0
                        acia      equ     $E000
                                  else
                        E004 acia equ     $E004
                                  endif
                                  endif

0000 F6   E004      outch ldb     acia
0003 C4   02              andb    #$02
0005 27   F9              beq     outch
0007 B7   E004            sta     acia
000A 39                   rts

                          end

0 Error(s) detected
```

Note that the first page of output is not shown.

IX. OBJECT CODE PRODUCTION

The object code output from the UniFLEX 6809 Assembler is a standard UniFLEX binary type file. This object code output can be turned on or off via the '+b' option on the calling line. When turned on, one of two types of object code can be produced: absolute and segmented. We shall discuss each type here and how they are produced.

Absolute Object Code Files

This is the most common type of object file for this assembler. It is the easiest type to understand and is what you will get by default. All that is required of the programmer to ensure his binary file will be the absolute type is to not use any of the segmentation directives described in the next paragraph. In this type of file, the binary is stored in a record format where each record has its own load address. This implies that an absolute binary file can load into memory at any point. If the object program is to operate under UniFLEX, however, it should load into low memory (generally starting at location 0000) with the stack in high memory as supplied by the system. With absolute object code files, code can be generated in any order. That is, you could assemble a section of code at $1000, then another at $0100, and finally another at $2400. This is done by using 'org' directives in the assembler.

Segmented Object Code Files

UniFLEX supports "segmentation" of binary files. It permits binary files to be broken into three segments of code, each having different characteristics. These three segments are called "text", "data", and "bss". Any code in a "text" segment is assumed by UniFLEX to be read-only. That is, UniFLEX will only read code in a "text" segment - it will not attempt to write into it. The "data" segment is sometimes referred to as "initialized data". It is code which has been produced by the assembler which can be either read or written. For example, one might put a temporary variable which required an initial value in this segment. At any point, the variable could be read or re-written. The "bss" segment is an area of reserved memory where no actual code has been produced by the assembler. It is sometimes referred to as "uninitialized data". The binary file does not contain any code to be placed in this section of memory, but rather only a size value for this segment. Its main purpose is to tell the operating system that memory is required in this area, but it does not need to be initialized to any values. The "bss" segment or area of memory can be read or written.

The reasons for breaking the binary file into these three sections is pretty clear. The "text" segment is known to be read-only. This implies the code will never be altered as long as the program runs. The operating system can make use of this fact by sharing this segment of memory in the event that more than one users wish to run the program at the same time. This can mean a considerable increase in efficiency of the system. The "data" section must be different for each user running the program. It is information (actual instructions or data) which must be initialized or loaded, but which can be altered at some later point. The "bss" segment really contains no code or data in the binary file. It is just a signal to the operating system that when the file is loaded it needs memory allocated in the area specified. The program should not assume that the memory in this segment will be initialized to any particular value.

This segmentation feature of UniFLEX binary files is really intended for programs produced by a relocating assembler with linking-loader. Though this assembler is not relocating, facilities have been added to allow the user to segment his object programs. If you have not fully understood the description of segmentation to this point, you are probably better off not attempting segmentation and instead using absolute binary files as described above. This assembler performs segmentation by maintaining three distinct location counters or program counters (PC's). At any point in the assembly, only one of these PC's is in effect. Any code generated by an instruction at that point is assembled at the address currently in the PC in effect. It is possible to switch to a different PC by use of one of the following three directives in the opcode field:

```
text
data
bss
```

Note that when the assembler is initiated it defaults to the "text" segment. It is possible to change which segment code is currently being generated into at any time (so long as the change is consistent with the forthcoming rules). In other words, you could begin with a "text" directive, enter 10 lines of code, then switch to the data segment with a "data" directive, enter 10 lines of code, then switch back to the text segment with another "text" directive, etc.

In general, each of the three sections should be a contiguous block of memory. To enforce this, the assembler only allows the programmer to set a particular segment's PC one time. After that the programmer must resume with the last address used for that PC. To specify an actual PC value, place an appropriate expression in the operand field as seen in the following examples:

```
text    $0000
data    $4000
```

To resume with the last address used by a particular segment, enter the instruction with no operand:

```
data
bss
```

Once the "text" and "data" segment PC's have been set, they cannot be changed. Any attempt to do so will result in an error. The "bss" segment is a different case. Since it does not contain any actual code, it is possible to change the current PC for the "bss" segment at any time, SO LONG AS THE NEW VALUE IS GREATER THAN THE OLD. The "bss" PC can be changed by either a "bss" statement with a value in the operand field or by an "org" statement.

If the first occurrence of a "text" or "data" directive does not have an address specified, the PC will default to $0000. If the first occurrence of a "bss" directive does not have an address specified, the last value of the "data" segment PC will be used.

It is not possible to generate code in a "bss" segment. Any attempt to do so will result in an error. By the same token, no instructions in "text" or "data" segments can cause a change in the PC without generating code or an error will result. This would be caused by an 'org' or 'rmb' statement.

When code is generated into the "data" segment it is actually written to a temporary file called '/tmp/asmbdataxxxxx' where xxxxx represents the current task number. At the end of the assembly this data is copied onto the end of the text code found in the main output file and the temporary file is immediately deleted.

Segmentation and UniFLEX Page Boundaries

Current versions of UniFLEX perform memory management via 4K pages. In order to share "text" segments among multiple users of a single program, the "text" segment must be in its own 4K page or pages. Thus the "data" segment must start on the first even 4K page address beyond the end of the "text" segment. If it does not, the assembler will issue a warning to that effect. It is up to the programmer to ensure that the first setting of the "data" segment PC is on an even 4K boundary. It is also required that the "text" segment begins at location $0000. If not, an appropriate warning is issued.

In the general case the "data" and "bss" segments are placed adjacent to one another. This is easily accomplished by not specifying the first "bss" address which forces it to the last one used by the "data" if the segments are in order in the source listing. The "bss" segment need not start on a 4K boundary.

The BASE Directive

There is one more directive related to PC's and segmentation which is sometimes very useful. It is the "base" directive. The "base" directive is used almost exactly like a segment PC directive (especially the "bss" segment) but serves a different purpose. It is really just an extra PC which can be set and maintained for the purpose of establishing offsets from some fixed address in an area outside the three segments. Generally it is used in conjunction with storing information on a stack. A short example program may be the best illustration:

```
stack   equ   $EF00

        base  $0000
temp    rmb   2
saved   rmb   2
junk    rmb   1

        text  $0000
start   ldu   #stack
        clra
        ldb   junk,u
        addd  temp,u
        std   saved,u
        end
```

As you can see, the "base" directive allowed us to setup the variables "temp", "saved", and "junk" which are offsets from the base location of the stack. The "base" directive does not actually create a segment - there is no such segment in UniFLEX. It merely sets up the new PC which can be temporarily used to establish offset variables. The "base" directive has most of the same permissions and restrictions as the "bss" segment except that it defaults to location $0000 if first called without an operand. No code may be generated while the "base" PC is in effect and 'org' statements or new "base" addresses are allowed.

Object Code File Permission Bits

If a program assembles with no errors and a transfer address has been supplied, the object code file will set the execute permission bits for the user and owner. If there are errors or no transfer address, neither user nor owner will have execute permission. If a "setid" directive was encountered during assembly, the "setid" permission bit will be set in the object file. In all cases the read and write permission bits will be set for both user and owner.

X. ERROR AND WARNING MESSAGES

The assembler supports warning messages and two types of error messages: fatal and non-fatal. A fatal error is one which will cause an immediate termination of the assembly such as a disk file read error. A non-fatal error results in an error message being inserted into the listing and some sort of default code being assembled if the error is in a code producing line. The assembly is allowed to continue on non-fatal errors. Warning messages are handled much like non-fatal errors: the message is inserted in the output listing and the assembly is allowed to continue. These warning messages may be suppressed by the 'W' option in the command line. Error messages may not be suppressed.

Most messages are output as English statements - not as error numbers. These messages announce violations of any of the rules and restrictions set forth in this manual and are, therefore, essentially self-explanatory. Non-fatal error messages are output with three asterisks preceding the message. Warning messages are output with two asterisks preceding the message. These can be used to quickly locate the messages either by eye or with an editor.

Fatal error messages are sent to the standard error channel. They are issued in the form:

```
<error message reported here>
Assembly aborted!
```

The messages which may come in the first line are listed later in this section.

Possible NON-FATAL error messages are as follows:

Illegal constant
: One of the numeric constants in the operand field was found to be illegal. This is generally caused by some digit or character which is not permitted in the number base in use.

Illegal indexed mode
: An error was discovered in the operand of an indexed mode instruction.

Illegal label
: The supplied label does not conform to the rules for labels as detailed in section III.

Illegal operand
: An error has been detected in the operand field.

Illegal option
: One of the options supplied in an 'opt' instruction is invalid. All options supplied after the one in error are ignored.

Illegal pc change
: An instruction was issued which attempted to illegally change the program counter. Examples are an "org" statement in a text or data segment and an "org" or other segment command that is lower than the current pc.

Macro exists
: An attempt was made to define a macro under a name which already exists. The assembler does not permit redefinition of macros.

Multiply defined symbol
: The label on the line in error is used as a label elsewhere in the program.

Not allowed in this context
: This message is generally associated with macros. Examples are operations that are illegal inside a macro or those which must be carried out inside a macro.

Not allowed in this segment
: The operation attempted is not permissible in the current segment type. Examples are an instruction that generates code placed in a bss or base segment and an "rmb" instruction in a text or data segment.

Operand overflow!
: This message is reported if too much data is generated by a single 'fcb', 'fdb', 'fcc', or 'sys' instruction. A maximum of 256 bytes of data can be generated by a single one of these instructions.

Phasing error detected
: This message is reported if the assembler detects an incongruity in addresses between pass one and two. Chances of this occurring are small, but the assembler will report such an error at the first detection. Phasing errors are only detected on lines which contain a label.

Relative branch too long
: A short branch instruction was used and the relative branch address is too large to fit in 8 bits.

Symbol table overflow
: The program being assembled contains too many symbols for the size symbol table in use. Increase the symbol table size via the command line option explained in section II.

Syntax error
: An error in some defined syntax has been detected. A common cause for this error is a label on an instruction which does not permit one or no label on an instruction which requires one.

Unbalanced clause
: One directive of a "clause" has been omitted such as an 'endif' with no 'if'. The assembler attempts to balance all such clauses and reports this error if an inconsistency is detected.

Undefined in pass 1
Some values must be defined in pass one in order for a two pass assembler to function properly. Examples are 'org' values, 'rmb' values, and symbols in conditional expressions.

Undefined symbol
One of the symbols in the line in error has not been defined.

Unrecognizable mnemonic or macro
The assembler was unable to find the string supplied in the opcode field in either the macro table or the mnemonic table.

Possible FATAL error messages are:

Device full - File: <filename>
When writing to the specifed file, the device on which it resides ran out of available space.

Error on standard output
An error was detected in writing to the standard output channel. If the assembled listing was not routed, the error was in writing to the terminal.

File does not exist - File: <filename>
The specified input file (either from the calling line or from a 'lib' statement) could not be found.

File is a directory - File: <filename>
The specified input file was a directory and not a text file.

Illegal option specified
One of the options supplied on the assembler calling line was invalid.

Illegal output file spec
The filename specified for the output binary file was too long.

Macro overflow!
One of the tables used in support of macros was overflowed. There are three such tables: the Macro Name Table where the names of all defined macros are stored, the Macro Activation Stack where certain information about each active macro is stored, and the Macro Data Table where the text of the macro is stored at definition time. These tables can be made larger by use of the "m" option on the assembler calling line. See the documentation of that option for details.

Missing directory - File: <filename>
One of the directory elements in the specified pathname did not exist.

No file specified
At least one input source file must be specified on the assembler calling line. None was found.

No permission - File: <filename>
The user was not allowed permission to the specified file.

Read error - File: <filename>
UniFLEX received an error in attempting to read the specified file.

Seek error - File: <filename>
UniFLEX detected an error when attempting a seek operation on the specified file.

UniFLEX error XX - File: <filename>
If an error is detected by UniFLEX which is not one of the preceding errors, this message will be printed where 'XX' represents the actual UniFLEX error number. The <filename> is the file in use when the error occurred.

Write error - File: <filename>
UniFLEX experienced an error in attempting to write to the specified file.

Possible warning messages are as follows:

Forced address truncated
This warning is printed if an address is forced to 8 bits (with the '<' character) and must be truncated to fit.

Illegal segmentation
This error occurs if one of the object code segmentation rules or guidelines is broken. Examples are not starting the "text" segment at location $0000, not starting the "data" segment on a 4K boundary, and a "text" segment that overlaps a "data" or "bss" segment.

Illogical forcing ignored
This warning is issued if address length forcing (with the '<' or the '>') is done in an operand where not possible. For example, the instruction ' LDB <[BUFCNT]' would result in such a warning.

Immediate value truncated
This warning occurs on lines where an immediate value must be truncated to fit in 8 bits. For example, ' LDB #$42E5' would result in such a warning.

As stated in a previous section, the total number of errors is reported at the end of the assembly and if warning messages are enabled, the number of warnings are also output.

NOTES: