# FLEX 6809
# Relocating Assembler

Technical Systems Consultants, Inc.

Table of Contents

# I. INTRODUCTION

This manual describes the operation and use of the 6809 Relocating Mnemonic Assembler for the 6809 FLEX™ Disk Operating System. The assembler accepts all standard Motorola mnemonics for the 6809 instruction set as well as all standard 6800 and 6801 mnemonics. Relocation, external references, 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 relocatable 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 FLEX.

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.

## Absolute Assemblers

An absolute assembler, such as the FLEX "ASMB", produces absolute machine code in which all addresses will be correct only if the program is loaded at the specified locations. For example, a program ORG'ed at $1000, contains a JMP instruction. At execution time, control will be transfered to the intended location by the JMP statement only if the program resides at $1000.

-------------------------------

FLEX is a trademark of Technical Systems Consultants, Inc.

Consider the following program:

```
                        ****
                        * cntchar - count the number of characters
                        *           in a null terminated string.
                        *
                        *           exit (b)=count of the characters

    1000                        ORG    $1000
    1000 8E   100D     CNTCHAR  LDX    #STRING   GET ADDR OF STRING
    1003 5F                     CLRB             SET COUNT TO ZERO
    1004 A6   80       LOOP     LDA    0,X+      GET A CHARACTER
    1006 27   04                BEQ    DONE      IF END OF STRING
    1008 5C                     INCB             COUNT THE CHARACTER
    1009 7E 1004                JMP    LOOP      GET NEXT CHARACTER

    100C 39            DONE     RTS              RETURN
    100D 48 4F 57 20   STRING   FCC    'HOW MANY CHARS ?',0
```

The program segment will work as intended only if it is loaded at $1000. If CNTCHAR was instead loaded at $2000, at the time the JMP instruction is encountered, control would transfer to $1004; however, we would intend for control to go to $2004.

Suppose it is not possible to know the starting location of a program segment. Such would be the case if each member of a team of programmers was responsible for developing a program segment. Since each segment is being developed independently of the others, it is not possible to know where one segment ends and, therefore, where the next is to begin.

## <u>Position</u> <u>Independence</u> <u>VS</u>. <u>Relocatability</u>

The 6809 instruction set offers one solution to this problem. PC relative addressing uses the Program Counter (PC) register as an index register in much the same manner as X,Y,U, or S is used as an index register. PC relative addressing generates. offs@ts (or indexes) based on the current PC value. Using this addressing mode, the program will work as intended regardless of its position in memory. We can make CNTCHR position independent by replacing the JMP instruction with BRA, and LDX immediate with LEAX (PC offset indexing). Our position independent version of CNTCHR would look like this:

```
                    ****
                    * cntchar - count the number of characters
                    *           in a null terminated string.
                    *
                    *           exit (b)=count of the characters

1000                        ORG    $1000
1000 30   8D 0009   CNTCHAR LEAX   STRING,PCR GET ADDR OF STRING
1004 5F                     CLRB             SET COUNT TO ZERO
1005 A6   80        LOOP    LDA    0,X+      GET A CHARACTER
1007 27   03                BEQ    DONE      IF END OF STRING
1009 5C                     INCB             COUNT THE CHARACTER
100A 20   F9                BRA    LOOP      GET NEXT CHARACTER
100C 39             DONE    RTS              RETURN
100D 48 4F 57 20    STRING  FCC    'HOW MANY CHARS ?',0
```

Notice that this CNTCHAR contains no references to absolute addresses. It will execute as intended regardless of where it is placed in memory. In general, however, position independent code contains more op code bytes and takes longer to execute than absolute code. Also, the programmer must make an effort not to use certain instructions, or restrict usage to particular addressing modes, so that the rules of postion independence are not violated. All code, on the other hand, can be considered "relocatable" when assembled using a Relocating Assembler.

## Relocating Assemblers and Loaders

Relocatable code is generated by a relocating assembler in such a way that addresses are not bound to absolute locations at "assembly time"; this binding of the address fields is deferred until "load time". A Relocating Loader binds the addresses at the time the program is brought into memory (loaded) to be executed. The binding or adjustment of the address fields is termed relocation. Because any code can be considered "relocatable", one can take advantage of the-speed and space savings of absolute code, with the adjustment of the address fields deferred until load time.

To illustrate how the Relocating Assembler and Linking-Loader work in conjunction with each other, we will present another version of CNTCHAR. Note that it does not contain any ORG statements, and that the code starts at location 0.

```
                          ****
                          * cntchar - count the number of characters
                          *           in a null terminated string.
                          *           exit (b)=count of the characters

 + 0000 8E     000D    CNTCHAR   LDX     #STRING   GET ADDR OF STRING
   0003 5F                       CLRB              SET COUNT TO ZERO
   0004 A6     80      LOOP      LDA     0,X+      GET A CHARACTER
   0006 27     04                BEQ     DONE      IF END OF STRING
   0008 5C                       INCB              COUNT THE CHARACTER
  +0009 7E     0004              JMP     LOOP      GET NEXT CHARACTER

   000C 39             DONE      RTS               RETURN
   000D 48  4F 57 20   STRING    FCC     'HOW MANY CHARS ?',0
```

A plus sign in front of an instruction indicates that the address field of that instruction must be increased by a "relocation constant". The "relocation constant" is the address at which the program is loaded for execution. For example, if the above program was loaded at $1000, the two relocatable address fields marked by "+" must each be increased by $1000 in order for the program to execute as intended. The address referred to by the first instruction would be $000D+$1000=$100D, which would now correctly point to STRING.

Address fields which do not require relocation are absolute addresses (their values remain the same regardless of the position of the program in memory). Since the loader does not have access to the source text, it cannot determine if an address field is absolute or relative (relocatable). In fact, it cannot distinguish addresses from data. Therefore, the assembler must indicate to the loader the address fields which require relocation. This communication is accomplished through "relocation records" which are appended to the object code file produced by the assembler. Such a file is called a "relocatable object code module".

## Linking-Loaders

As discussed previously, it is necessary or desirable for parts of a program (modules) to be developed separately. Each module must be assembled and tested separately prior to final merging of all the modules. During this merging process of the modules, it is necessary to resolve references in a module which refer to addresses or data defined in another. The resolution of "external references" is called linking. The assembler must provide information to the linker, in a manner similar to relocation records, concerning the address fields which must be resolved.

Since the production of relocation records and external records are similar processes, both of these features are often included in a relocating assembler. A linking-loader is a program which can take the output of a relocating assembler (object code, relocation records, and external records) and produce executable object code. Linking-loaders, therefore, can perform the following tasks:

1) load a relocatable object code module into memory,
2) relocate any relative address fields,
3) resolve any external references, and
4) optionally execute the resultant program.

At this time a few definitions would be in order. A global symbol is a location within a module that can be referenced from other separate modules. Each global symbol has a unique name associated with it. The linking-loader is given a list of the global symbols in each module. An external reference is a reference in one object module to a global symbol in another object module. And finally, a common block is a block that can be declared by more than one object module. The data used in the common blocks is "common" to all of the modules that have declared the common block.

# II. GETTING THE SYSTEM STARTED

The FLEX 6809 Relocating Mnemonic Assembler is very simple to use. 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 FLEX 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) 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:

```
+++RELASMB,<filename>
```

The three plus signs are FLEX's ready prompt, RELASMB is the name of the relocating assembler file (it has a .CMD extension), and the <filename> is the standard FLEX specification for the source file you wish to assemble. The <filename> defaults to a .TXT extension and to the assigned working drive if an explicit extension and drive number are not given. In this and forthcoming example command lines, a comma is used to separate items. It is also possible to use a space or spaces in this capacity.

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 the two associated with output. We say two because 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.

The object code can be in the form of a relocatable binary disk file or no object code output at all. Since no specifications are made concerning object code output in the above example, the assembler will assume the default case which is a relocatable 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 a .REL extension. If such a file already exists, you will be asked:

    Delete Old Binary (Y-N)?

to which you may respond 'Y' which will delete the existing file and continue to create the new file or 'N' which will immediately terminate the assembly, returning to FLEX with the old binary file remaining intact.

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

    +++RELASMB,<input file spec>[<binary file spec>][,+<option list>]

This binary file specification will default to a .REL extension and to the assigned working drive. If a file by that name already exists on the specified drive, you will be prompted as described above.

A temporary scratch file is created to hold the external and relocation records that are produced during assembly. This file is only created if a binary file is being produced. The name of this file will be the name of the source input file with a .SCR extension. If this file should happen to exist, the assembler will prompt the user:

    Delete Old Scratch File (Y-N)?

A response of N will abort the assembly. Normally, this file will be automatically created and deleted by the assembler. This scratch file will exist on the same drive as the binary output file. Please ensure that there is room on the disk for the binary output file, this temporary scratch file and the cross-reference information file if it is selected.

## 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:

    +++RELASMB,<input file spec>,<binary file spec>][,+<option list>]

The square brackets indicate that the binary file spec and the option list are optional. The plus sign is required to separate the option list from the file specifications. The <option list> is a set of single character flags which either enable or disable a particular option. 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:

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

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.

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 FLEX and prints it in the header. This option causes the date to be omitted.

F Enable debug or Fix mode. There are two forms of line comments. One begins with an asterisk (*) the other with a semicolon (;), both in the first column of the source line. If the comment begins with a semicolon, the F option will instruct the assembler to ignore the semicolon and process the line as though the semicolon never existed. The asterisk in the first column of a source line will always denote a comment regardless of the state of this option.

G Turns off printing of multiple line object code instructions. Certain directives (FCB, FDC, and FCC) can produce several lines of output listing for only one instruction line. This option prints the first line of output from such an instruction (the line which contains the source) but suppresses the printing of the subsequent lines which contain only object code information.

I Enable the appending of all internal symbols. Normally, the relocating assembler will append only the global symbol table information to each binary output module. This is then used by the linking-loader to resolve externals. By specifying this option, all of a source module's symbol table information is appended to its binary module. The one exception to this is symbol table information for external symbols; this is never written out in the symbol table section of the binary modules. This will be useful in symbolic debugging.

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.

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.

O Limit all symbols to only six characters internally. Normally, the assembler will allow the user to define and use symbols that contain eight characters. However, in some cases it may be necessary to limit the uniqueness of the symbols to only six characters to conform to the absolute assembler, "ASMB". For example, if you define a symbol, "TESTCASE", and use it in the operand field as "TESTCA" since the absolute assembler, "ASMB", only used six characters, the O option would be necessary for assembly.

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 itself.

U Set all undefined symbols as external. In some cases the user may wish to assemble a module that has some undefined references in it without specifying these as external symbols. The U option will treat all undefined references as external references. The U option should not substitute for the good programming pratice of listing all external symbols in the operand field of the "EXT" directive.

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.

X Produce cross-reference information. The assembler has the ability of producing an output file containing information that can be used by a cross-reference program to create a cross-reference listing. This file will be called the name of the source input file with a .REF extension and will appear on the same disk as the input source file. If a cross-reference file exists, the user will be prompted for the file's deletion prior to assembly.

Y This option overrides the prompt for deleting an existing binary file. In other words, if the Y option (stands for YES) is specified, an existing binary file of the same name as the one to be created will be automatically deleted without a prompt. If this option is selected, both the scratch file and cross-reference file will be automatically deleted if they previously existed.

P<no This option allows the programmer to specify a page number at which to start printing of the assembled listing. Complete instructions can be found in the next paragraph.

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 escape/return sequence found in FLEX. The desired page number is specified along with the 'P' option in the option list. It must directly follow the 'P' and should be a decimal number from 1 to 65,536. The page number itself must be directly followed by a non-alphanumeric character (terminator) such as a comma or space. This implies that the 'P' option, if specified, MUST BE THE LAST OPTION SPECIFIED. The page number may also be followed or terminated by a plus sign as used in the following paragraph. 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 turned on by default.

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 in the manner shown here:

```
+++RELASMB,<in file>,<bin file>,+<options>,+<prm.1>,<prm.2,<prm.3>
```

The parameters are optional but must be separated from the rest of the line by the second plus sign and from each other by commas. As stated, these parameters are simply strings of characters. The string of characters which makes up an individual command line parameter can not have spaces or commas iinbedded in it. Note that if one wishes to enter a command line parameter but no options, he must still place both plus signs in the command as seen in this example command line:

```
+++RELASMB,ANYFILE,++PARAMETER1,PARAMETER2
```

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.

## 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 FLEX. To do so one must use the 'P' command provided with FLEX. This 'P' command reads a printer driver file which you can supply to output to any hardcopy device you want. It effectively switches the output of the assembler from going to the terminal to going to the printer. For example:

```
+++P,RELASMB,TESTFILE
```

would cause the assembled listing of the source file TESTFILE.TXT to be output to the printer. For further details of use of the 'P' command refer to the FLEX User's Manual and Advanced Programmer's Guide.

EXAMPLES:

RELASMB,TEST

Assembles a file called TEST.TXT on the assigned working drive and creates a binary file called TEST.REL on the same drive. The assembled listing is output to the terminal as is the symbol table.

RELASMB,TEST,+LS

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

RELASMB,0.TEST,1.TEST.TST,+FLSY

Assembles a file from drive 0 called TEST.TXT and produces a binary file on drive 1 called TEST.TST. The source file will have all lines starting with a semicolon assembled as though the semicolon did not exist. No listing or symbol table is output and if a file by the name of TEST.TST already resides on drive 1, it will be automatically deleted before the assembly starts.

1.RELASMB,0.TEST.BAK,+BNGS0

Loads the assembler itself from drive 1 and assembles a file called TEST.BAK found on drive 0. No binary file is produced. The assembled listing is output with line numbers turned on and multiple line generated code turned off. No symbol table is printed; all symbols are unique to only six characters.

RELASMB,0.TEST.BAK,+GS0BNP26

This command performs just like the last with two exceptions. First, the assembler itself is loaded from the whatever the assigned system drive is. Second, the assembled listing output does not begin until the assembler reaches page number 26.

0.RELASMB,0.RELASMB,0.RELASMB,+GW,+MINI,'ASSEMBLER FOR 5" DISK'

This command looks a little confusing, but it was done to accentuate the method in which default extensions work. The assembler itself (a file called RELASMB.CMD) is loaded from drive 0, the file called RELASMB.TXT found on drive 0 is assembled, and a binary file is produced on drive 0 by the name RELASMB.REL. Note that it was not necessary to specify the binary file name in this case since RELASMB.REL is what the default would have been. The assembled listing is output with multiple line code generation suppressed and warning messages suppressed. There are two parameters which may be passed into the source listing. The first is the single word 'MINI'. The second parameter is the entire string, 'ASSEMBLER FOR 5" DISK', excluding the single quote delimiters (everything starting with the A and ending with the K).

# III. ASSEMBLER OPERATION & SOURCE LINE COMPONENTS

The FLEX Relocating 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 1F hex) other than the carriage return ($0D) 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) or semicolon (hex 3B) 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.

a) Ordinary Labels
   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 8 characters are significant,
   5) The label field must be terminated by a space or a return.

b) Local Labels

   1) Local labels follow many of the same rules as ordinary labels. They must begin in column one and they must be terminated by a space or a return.
   2) Local labels are comprised of a number from 0 to 99. These numbers may be repeated as often as desired in the Same source module; they need not be in numerical order.
   3) Local labels may be treated as ordinary labels; however, they cannot be global or external. They may not be used in the label field of an 'equ' or 'set' directive.
   4) Local labels are referenced by using the local label number terminated with an 'F' for the first forward reference found or a 'B' for the first backward reference found. Realize, a backward or forward reference can never refer to the same line that it is found on. For example,

```
2     BEQ   2F    "2F" = next occurance of "2"
2     LEAX  1,X   both branches point here!
2     BRA   2B    "2B" = previous occurance of "2"
```

5) Local labels should be used primarily (but not necessarily exclusively) for branching or jumping around some sections of code. In most cases, branching around a few lines of code does not warrant the use of an ordinary label. When making reference to a nearby location in the program there often is no appropriate name with much significance; therefore, programmers have tended to use symbols like L1, L2, etc. This can lead to the danger of using the same label twice. Local labels have freed the programmer from the necessity of thinking of a symbolic name of a location. For example,

```
        BEQ    29F
        LEAX   1,X
29      STX    0,Y
```

   Furthermore, local labels only require four bytes to store internally; ordinary labels require twelve bytes each. The maximum of local labels is dependent upon the available memory.

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 interchangeably.
2) This field must be terminated by a space if there is an operand or by a space 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.
2) This field is terminated with-a space 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.
3) Comments may contain any characters from SPACE (hex 20) thru DELETE (hex 7F).
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 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, constant, relocatable or external value may be used in expressions. As described above under the label field, a label is comprised of letters, digits, and underscores beginning with a letter. The label may be of any length, but only the first 8 characters are significant. Any label used in the operand field must be defined elsewhere in the program. Local labels may also be used in the operand field. 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. In a relocatable module, the value of the PC designator is relocatable; its value is given at loading time.

## TYPES OF EXPRESSIONS

There are three types of expressions possible in the FLEX Relocating Assembler. These include absolute, relocatable and external expressions.

a) Absolute Expressions

An expression is absolute if its value is unaffected by program relocation or the module is considered absolute by the presence of an ABS and ORG directive. An expression can be absolute, even though it contains relocatable symbols, under both of the following conditions:

1) The expression contains an even number of relocatable elements
2) The relocatable elements must cancel each other. That is, each relocatable element (or multiple) must be canceled by another relocatable element (or multiple). In other words, pairs of relocatable elements must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

For example, rel1 and rel2 are two relocatable symbols in a module; the following examples are absolute expressions.

```
rel1-rel2
5*(rel1-rel2)
-rel1+3*(50/4)+(rel2-4)
```

b) Relocatable Expressions

An expression is relocatable if its value is affected by program relocation in a relocatable module. A relocatable expression consists of a single relocatable symbol or, under all three following conditions, a combination of relocatable and absolute elements.

1) The expression does not contain an even number of relocatable elements.
2) All the relocatable elements but one must be organized in pairs that cancel each other. That is, for all but one, each relocatable element (or multiple) must be canceled by another relocatable element (or multiple). The elements that form a pair need not be contiguous in the expression.
3) The uncanceled element can have either positive or negative relocation.

For example, rel1 and rel2 are relocatable symbols from a module; the following examples are relocatable:

```
-rel2+3*5          negative relocation
rel1+3
rel1-(rel2-rel1)
*+1                PC Designator
```

c) External Expressions

An expression is external if its value depends upon the value of a symbol defined outside of the current source module. Either an external expression consists of a single external symbol or under both of the following conditions, an external expression may consist of an external symbol, relocatable elements and absolute elements:

1) The expression contains an even number of relocatable elements
2) The relocatable elements must cancel each other. That is, each relocatable element (or multiple) in a module must be canceled by another relocatable element. In other words, pairs of relocatable elements must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

For example, ext1 is an external symbol, rel1, rel2 have the same meaning as above in the previous examples; the following examples are external:

```
rel1-rel2+ext1-rel2+rel1
5+ext1-3
3/(rel2-rel1)-ext1
```

## 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. Realize that many of these operators will only apply to absolute symbols and expressions. It does not make sense to multiply a relocatable or external value at assembly-time! Only the + and - operators can apply to relocatable and external symbols and expressions; all of the other operators can be used with absolute symbols and expressions.

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 will be all ones. If false, the resulting value 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. Note that only long branches are allowed for external expressions. 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. Realize that direct addressing does not make sense for relocatable or external expressions; the normal default will be extended.
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. If the module is relocatable, PC Relative offsets will be calculated at assembly time. Any PC Relative addressing to externals will always reserve a 16 bit address. PC Relative Addressing is distinguished from normal PC Offset Indexing by the use cf '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 for absolute modules and extended addressing for relocatable modules and external references. 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 in an absolute module:

```
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. Relocatable modules will use extended addressing as the normal default.

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. In some cases, such as PC Relative addressing, the forcing of direct addressing for relocatable and external expressions will be ignored. 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
           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
           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
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

SW12 — 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
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 relocating 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 three types of directives in this assembler, those associated with macros, those associated with conditional assembly, and those which generally can be used anywhere which we shall call "standard directives". This section is devoted to descriptions of these directives which are briefly listed here:

| | | |
|---|---|---|
| ABS | SET | RPT |
| ORG | REG | LIB |
| END | SETDP | GLOBAL |
| RMB | PAG | DEFINE and ENDDEF |
| RZB | SPC | EXT |
| FCB | TTL | NAME |
| FDB | STTL | COMMON and ENDCOM |
| FCC | ERR | OPT EQU |

Descriptions of each directive and its use follow.

## ABS

The ABS or Absolute directive signals the relocating assembler to assemble the following module as an absolute module. This means that there will be no relocatable expressions. The ABS directive should appear in the source module before any other code in the source module. Its syntax is as follows:

```
ABS
```

There should be no label or operand. Use the ABS directive when a module has to be loaded at a particular location; it is not to be assigned a load address by the linking loader.

## 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. The ORG directive should be used only in absolute modules; they are ignored and flagged as errors in relocatable modules. All expressions in absolute modules are absolute or external.

## END

The END pseudo-op is used to signal the assembler that the end of the source input module has occurred. This terminates whatever pass is currently being executed. 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 relocatable binary file. It is optional, and if supplied when no binary file is being produced, 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 and separating modules.

## 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 locations and therefore the contents are undefined at run time. The proper useage is shown here:

```
[<label>] RMB   <absolute expression>
```

The label is optional, and the expression is a 16 bit quantity. The RMB is treated as a RZB in relocatable modules except in common blocks.

## RZB

The RZB or Reserve Zero 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 <absolute expression>
```

where the absolute 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 absolute, relocatable or external 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 quantities 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 <nonexternal expression>
```

The label is strictly required in equate statements. Absolute or relocatable expressions are allowed; external expressions are illegal. If the expression is relocatable both the value and relocatable attribute will be assigned to the label.

## 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 <non external 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 occurances 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 [<absolute 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). Direct addressing has more meaning for an absolute module.

## 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 have been previously selected 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 [<absolute expression>]
```

Where the absolute 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. Both the space count and keep count must be absolute expressions.

## TTL

The TTL directive allows the user to specify a title for 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>
```

All the text following the TTL 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 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.

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  <absolute 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 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 FLEX file specification. The default drive is the assigned working drive and the default extension is .TXT. 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. If nested, LIB files must come from the same drive.

## GLOBAL

The GLOBAL directive is used in relocatable modules to inform the assembler that the symbols declared global should be passed on to the linking-loader. The syntax of the GLOBAL directive is:

```
    GLOBAL <label1>[,<label2>, . . .]
```

Where label1, label2, etc. represent the symbolic names of the labels to be declared as global; each label should be separated by a comma. Realize, the GLOBAL directives must occur before the use or definition of the symbol. Normally, declare global symbols at the beginning of the source module. Macros and local labels cannot be declared global.

## DEFINE and ENDDEF

These are convenience directives that work much the way GLOBAL works. The DEFINE directive informs the assembler that all labels declared in the label field will also be declared as global symbols. This "define" mode will be in effect until an ENDDEF directive is encountered. For example,

```
         DEFINE
TEMP1    FDB      0,$FFFF
START    LDX      1
         ENDDEF
```

This example simply defines the two labels TEMP1 and START as global. This directive works well when many symbols must be declared as global while they are initialized to various values.

## EXT

The EXT directive declares symbols to be external to this particular module. Macros and local labels cannot be declared external. The syntax of the EXT directive is:

```
EXT  <label1>[,<label2>, . . .]
```

Where label1, label2, etc. is an ordinary label as in GLOBAL; each label, should be separated by a comma. When the assembler comes across a label declared external in the operand field, external records will be written out to the binary output module. As with the GLOBAL directive, the EXT directive should appear before the actual use of the external symbol; usually at the beginning of the source module. These external records will be used by the linking-loader.

## NAME

Each binary output module can be given a module name with the NAME directive. The module name is used by the linking-loader in reporting errors and address information. it is strongly recommended to give each module a name. The syntax of the NAME directive is:

```
NAME <name of the module>
```

The name of the module can be a maximum of 8 characters. If more than one NAME directive occurs in the source module, the last name given will be the name given to the module.

## COMMON and ENDCOM

It is possible to establish common blocks in the relocating assembler. These can only be named and uninitialized common blocks.

```
<name> COMMON
```

A common block declaration is terminated by the use of the ENDCOM directive. The only directive allowed between the COMMON and ENDCOM is RMB. The RMB directives define the size of the common block; labels may be associated with each RMB within the common block. For example,

```
TEST        COMMON
TEMP1       RMB         10
TEMP2       RMB         5
            ENDCOM
```

This common block named TEST has two variables, TEMPl and TEMP2, associated with it and is of length 15 bytes. A common block and its variables are considered external by the assembler. Only one common block of a particular name should appear in a module; however, up to 127 common blocks of different names may occur in a module. As common blocks are treated as externals, the linking-loader handles the resolution of references to the common blocks automatically. For example,

```
TEST        COMMON
TEMP1       RMB       10
TEMP2       RMB       5
            ENDCOM
START       LDD       TEMP1
            . . .
```

Common blocks are very useful for passing parameters and keeping common information around. Furthermore, the common block will have the name of the common block as its module name; this is done automatically by the assembler. Common blocks must be accompanied by executable code in the same module; that is, a common block cannot be the only item in a single source module.

## 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 may be imbedded in the option list. The options are set during pass two only. 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 last option may be used to 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  <absolute expression>
            .
            .    conditional code goes here
            .
        ENDIF
```

The ENDIF requires no additional information but the IF requires an absolute 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  <absolute 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   <absolute 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 tvio 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.

<u>IMPORTANT</u> <u>NOTE</u>:
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   <absolute 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.

<u>IMPORTANT</u> <u>NOTE</u>

In order for conditionals to function properly, they must be 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 directive 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. Macros cannot be global or external! 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 deliniiter 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 most 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    COUNTd,-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.

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. Of course, the easiest way to do this is by using local labels within the macro.

## 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 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 redefined.

# VIII. SPECIAL FEATURES

This section covers a few special features of the 6809 Relocating 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, four items of information may be printed: the total number of errors encountered, the total number of warnings encountered, the total number of excessive branches or jumps 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
```

Excessive branches or jumps are printed after the error count and after the warning count. If no warnings were reported, the excessive branch/jump count will be displayed after the error count.

```
0 Error(s) Detected    3 Excessive BRANCH/JUMP(S) Detected
```

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 si'gn 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 10, all opcodes beginning in column 17, and all comments beginning in column 27 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.

## Fix Mode

The relocating assembler has the capability to accept comment lines that begin with either an asterisk (*) or a semicolon (;). If a semicolon is used, the F option of the assembler will assume that the "comment" is a valid instruction to be assembled. Therefore, the assembler will act as though the semicolon did not exist at all; the rest of the information on that line will be assembled. For example,

```
;LABEL1 LDX 2
; LDD TEST,U
```

With the F option invoked, these two lines will be normally assembled. This aides in the debugging process.

## Local Labels

Local labels are available in the assembler. These local labels allow the programmer to reuse labels over; in this way meaningless labels can be substituted with local labels. See section three, ASSEMBLER OPERATION & SOURCE LINE COMPONENTS, under the description of the label field for more information on local labels.

## Auto Table Setup

The assembler automatically sets up all necessary tables and buffers depending on the amount of memory contained in your system. The amount of memory contained is determined by reading the memory end value (MEMEND) in FLEX. The tables or buffers which can vary in size are as follows:

SYMBOL TABLE

This table is where all symbols are maintained during assembly. Each entry into the symbol table requires 12 bytes, 8 for the name, two for the address or value associated with it and two for flags and common block information. This assembler uses a hashed symbol table technique which can mean that symbols are pseudo-randomly scattered throughout the table. This method is very fast but has one drawback - the assembler is usually not able to completely fill all positions in the table. Thus a symbol table full error message does not necessarily mean that every position in the table is really full. It simply means the assembler was unable to put any more symbols in the table due to the hashing technique used. However, if the number of 12 byte slots is a prime number, the hashing technique used will fill every slot in the table and examine each slot only once. The larger the table, the faster the hashing method will execute and accordingly, the faster the assembler will run. If possible, it is good to have 50% more space in the symbol table than will be required to actually hold the number of symbols in a program.

LOCAL LABEL LIST

This list contains all of the local labels sequentially encountered in the source code. Local labels are entered and searched sequentially. Each local label requires 4 bytes, 1 byte for the representation of the local label, 2 bytes for the address and 1 byte for flags. This list should contain enough locations for all of the local labels used in a single source module.

SOURCE BUFFER

This is the buffer which holds the source program to be assembled. The assembler reads only one line of source code at a time. The assembly operation is carried out on each line individually. This buffer may be any size so long as it will hold the longest line that may be encountered in the source.

MACRO TEXT BUFFER

This is the buffer where macros are stored upon definition. If few macros are to be used, it can be very small. The more macros anticipated, the larger the buffer should be.

MACRO NAME TABLE

This is a table where macro names and locations are stored. Each entry requires 12 bytes, 8 for the macro name, 2 for the address of the macro body in the macro text buffer and 2 for flags. Again, if few macros are to be defined, this table can be quite small.

MACRO ACTIVATION BUFFER

This buffer is an area where the macro processor builds an activation stack for all macros currently under execution. Each time a macro is called, certain information about it is placed on this stack. If no parameters are supplied on the calling line, an entry on the stack requires some 10 bytes. Any calling line parameters will raise this amount. If macros are not nested , only one entry will be on the stack at a time. However, if one macro calls another, there must be two entries on the stack and so on. The minimum size of the stack necessary depends on tne amount of macro nesting done.

The assembler attempts to provide a general sizing of these tables for any size system in which it is run. This sizing is done according to the following approximate formulae:

```
Let AVM = MEMEND - MEMBEG
          where AVM implies available memory
                MEMEND refers to the end of memory as in FLEX
                MEMBEG is the start of the RAM buffer area
Then

SYMBOL TABLE = (AVM-256 bytes)*0.5
LOCAL LABEL LIST = (AVM-256 bytes)*0.25
MACRO TEXT BUFFER = (AVM-256 bytes)*0.1875
MACRO NAME TABLE = (AVM-256 bytes)*0.03125
MACRO ACTIVATION BUFFER = (AVM-256 bytes)*0.03125
SOURCE BUFFER = AVM-(sum of above spaces)
```

These table sizes can be set manually by the programmer if so desired. See the section on adapting to your system for details.

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:

```
+++RELASMB,<filename>,+<options>,+<prm.A>,<prm.B>,<prm.C>
```

As can be seen, the three parameters are separated by commas.

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   PAG,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:

```
+++RELASMB, FILE, +BGDS, +1
```

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

```
OUTPUT ROUTINE FOR PORT #1             TSC 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. 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 not enough memory. 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.

All 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. Error messages are output with three asterisks preceding the message. Warning messages are output with two asterisks preceding the message. This can be used to quickly locate the messages either by eye or with an editor.

Possible NON-FATAL error messages are as follows:

EXTERNAL SYMBOL NOT ALLOWED IN THIS CONTEXT
EXTERNALLY DEFINED SYMBOL USED INTERNALLY
GLOBAL SYMBOL NOT DEFINED IN THIS MODULE
ILLEGAL CONSTANT
ILLEGAL INDEXED MODE
ILLEGAL LABEL
ILLEGAL OPERAND
ILLEGAL OPTION
ILLEGAL RELOCATABLE EXPRESSION
LOCAL LABEL TABLE OVERFLOW
MACRO EXISTS
MULTIPLY DEFINED EXTERNAL SYMBOL
MULTIPLY DEFINED SYMBOL
NESTED COMMON BLOCKS NOT ALLOWED
NOT ALLOWED IN THIS CONTEXT
ONLY ABSOLUTE SYMBOLS ALLOWED IN THIS CONTEXT
ONLY ONE EXTERNAL SYMBOL ALLOWED IN THIS CONTEXT
OPERAND OVERFLOW!
OVERFLOW!
PHASING ERROR DETECTED
RELATIVE BRANCH TOO LONG
SHORT BRANCHING TO EXTERNAL LABELS NOT ALLOWED
SYMBOL TABLE OVERFLOW
SYNTAX ERROR
TOO MANY COMMON BLOCKS!
UNBALANCED CLAUSE
UNDEFINED IN PASS 1

UNRECOGNIZABLE MNEMONIC OR MACRO
UNDEFINED SYMBOL

A couple of these could use some elaboration. The OPERAND OVERFLOW message results from attempting to generate too much data from a single FCB, FDB, or FCC instruction. A maximum of 256 bytes of data can be generated by a single instruction of that type. An example of the OVERFLOW error is when LIB files are nested more than 12 levels deep. The UNBALANCED CLAUSE message results when one directive of a clause is missing such as an ENDIF with no IF. The NOT ALLOWED IN THIS CONTEXT message is generally associated with macros, ie. when an operation is attempted that is illegal inside a macro or vice versa. The PHASING ERROR DETECTED 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.

Possible FATAL error messages are:

ILLEGAL FILE NAME
NO SUCH FILE
ILLEGAL OPTION SPECIFIED
INSUFFICIENT MEMORY
MACRO OVERFLOW!

The first three are associated with errors in the command line. The INSUFFICIENT MEMORY message is issued when automatic table setup is enabled and there is not at least 3K of buffer area. The MACRO OVERFLOW message occurs when any of the macro buffers is overflowed.

Possible warning messages are as follows:

FORCED ADDRESS TRUNCATED
IMMEDIATE VALUE TRUNCATED
ILLOGICAL FORCING IGNORED
NO PRECEDING ABS--ORG IGNORED

The first warning is printed if an address is forced to 8 bits (with the '<' character) and must be truncated to fit. The second 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. The third 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. Finally, the fourth occurs when an ORG is put into code without first informing the assembler that the code is to be absolute with an ABS directive.

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.

# X . ADAPTING TO YOUR SYSTEM

In general, if you have FLEX up and running and have at least 16K of user memory, there will be no need for any adapting whatsoever. Some users, however, always feel the need to modify certain things and this section describes how to make all the changes which we felt might be desired (and which would be feasible). With the exception of MEMEND, these changes must all be made to the object code of the assembler. This can be done by loading the assembler (with a GET RELASMB.CMD command), making the desired changes in memory, ard saving the result on disk (with the SAVE command).

## MEMEND

If using the automatic table setup which the assembler performs for you, the MEMEND value in FLEX should be set to the last address which the assembler should use. FLEX initializes MEMEND to the actual end of user memory. You may have an application where you don't want this to be the case. If so, set MEMEND manually before executing the assembler.

## EXIT ADDRESS

When the assembler is finished, it jumps into FLEX's warm start. If you wish it to exit to some other address, place that address at location $02FC.

## OUTPUT CHARACTER ROUTINE

As supplied, the assembler outputs through FLEX's PUTCHR routine. This takes advantage of the escape-return sequence in FLEX to pause or terminate an output Tisting. If, however, you wish to output through some user supplied output routine, place the address of such a routine at location $02FF. This routine should output the character in the A register and return without affecting any registers except the condition codes.

## LINES PER PAGE

If the page mode is selected, the assembler automatically places a header at the top of each page (with the current page number) and performs a page eject at the bottom. The number of lines which are output before the page eject occurs (including the header lines) is set at location $001C in the assembler. This value is currently set to 58 lines. Change if desired.

## LABELS PER LINE

The symbol table output is done with multiple symbols per line. Currently there are 5 labels per line. If desired, this number can be changed by setting the byte at location $001D.

## AUTO FIELDING COLUMNS

As explained in the special features section, this assembler performs auto fielding. The columns in which the fields are placed are currently set to 10, 17, and 27 for the mnemonic, operand, and comment fields respectively. If desired, these columns can be changed by altering the three bytes beginning at location $001E.

## SETTING UP THE TABLES MANUALLY

If you wish to set up the necessary tables and buffers manually, you may do so. This requires that you supply the starting and ending addresses of the five tables described in the section on special features and set a byte called MANUAL to some non-zero value. This MANUAL byte is located at $0003 and is currently equal to zero. The assembler tests this byte and if zero, performs automatic table setup. If non-zero, the assembler picks up the table start and end addresses from the following locations:

| | |
|---|---|
| SOURCE BUFFER BEGIN | $0004 |
| SOURCE BUFFER END | $0006 |
| MACRO NAME TABLE BEGIN | $0008 |
| MACRO NAME TABLE END | $000A |
| MACRO ACTIVATION BUFFER BEGIN | $000C |
| MACRO ACTIVATION BUFFER END | $000E |
| MACRO TEXT BUFFER BEGIN | $0010 |
| MACRO TEXT BUFFER END | $0012 |
| LOCAL LABEL LIST BEGIN | $0014 |
| LOCAL LABEL LIST END | $0016 |
| SYMBOL TABLE BEGIN | $0018 |

SYMBOL TABLE END $001A

NOTES:

1) The Macro Name Table and the Symbol Table must be an even multiple of 12 bytes in size!

2) The assembler uses a large stack area to maintain all its temporary variables and buffers. This stack requires 0AD0 hex bytes of RAM starting at FLEX's MEMEND and growing down from there. This implies two things: if manually setting up the tables you must leave this space free and if you want to move where the stack resides, you must set MEMEND accordingly as the assembler will always place its stack so that it sets up against MEMEND.

3) When manually setting up the tables, you will probably want to use all the space between the end of the assembler itself and (MEMEND-$0AD0). Thus the lowest table would start just after the last byte of the assembler as found on the disk and the highest table would end at (MEMEND-$0AD1).

4) The tables must be put into memory in the exact same order as they are listed above. They must be put into contiguous memory.

# XI. APPENDIX A: LIST OF PRIME NUMBERS FROM 773 TO 2239

The symbol table is a large table that contains information,about each symbol used in the source module. Each symbol in this table is addressed by using a hash function. This hash function uses the characters that make up the symbol and yield an address into the symbol table. It is possible that two symbols will yield the same address in the table. When this happens the second symbol is put into another slot in the table based upon its initial hash value. Therefore, it is possible that this table is not quite full, yet the assembler will report a symbol table overflow message. The hash algorithm is such that it is optimal when the table size is a prime number of slots. Included is a table of prime numbers from 773 to 2239.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 773 | 787 | 797 | 809 | 811 | 821 | 823 | 827 | 829 | 839 |
| 853 | 857 | 859 | 863 | 877 | 881 | 883 | 887 | 907 | 911 |
| 919 | 929 | 937 | 941 | 947 | 953 | 967 | 971 | 977 | 983 |
| 991 | 997 | 1009 | 1013 | 1019 | 1021 | 1031 | 1033 | 1039 | 1049 |
| 1051 | 1061 | 1063 | 1069 | 1087 | 1091 | 1093 | 1097 | 1103 | 1109 |
| 1117 | 1123 | 1129 | 1151 | 1153 | 1163 | 1171 | 1181 | 1187 | 1193 |
| 1201 | 1213 | 1217 | 1223 | 1229 | 1231 | 1237 | 1249 | 1259 | 1277 |
| 1279 | 1283 | 1289 | 1291 | 1297 | 1301 | 1303 | 1307 | 1319 | 1321 |
| 1327 | 1361 | 1367 | 1373 | 1381 | 1399 | 1409 | 1423 | 1427 | 1429 |
| 1433 | 1439 | 1447 | 1451 | 1453 | 1459 | 1471 | 1481 | 1483 | 1487 |
| 1489 | 1493 | 1499 | 1511 | 1523 | 1531 | 1543 | 1549 | 1553 | 1559 |
| 1567 | 1571 | 1579 | 1583 | 1597 | 1601 | 1607 | 1609 | 1613 | 1619 |
| 1621 | 1627 | 1637 | 1657 | 1663 | 1667 | 1669 | 1693 | 1697 | 1699 |
| 1709 | 1721 | 1723 | 1733 | 1741 | 1747 | 1753 | 1759 | 1777 | 1783 |
| 1787 | 1789 | 1801 | 1811 | 1823 | 1831 | 1847 | 1861 | 1867 | 1871 |
| 1873 | 1877 | 1879 | 1889 | 1901 | 1907 | 1913 | 1931 | 1933 | 1949 |
| 1951 | 1973 | 1979 | 1987 | 1993 | 1997 | 1999 | 2003 | 2011 | 2017 |
| 2027 | 2029 | 2039 | 2053 | 2063 | 2069 | 2081 | 2083 | 2087 | 2089 |
| 2099 | 2111 | 2113 | 2129 | 2131 | 2137 | 2141 | 2143 | 2153 | 2161 |
| 2179 | 2203 | 2207 | 2213 | 2221 | 2237 | 2239 | | | |

NOTES: