```
/*
      HEADER:           CUG149;
      TITLE:            6801 Cross-Assembler (Portable);
      FILENAME:   A68.DOC;
      VERSION:    3.5;
      DATE:       08/27/1988;
                  update Nov 2011, compiled for lcc-32 windows by HRJ
                  update Feb 2019 V3.5+, added S-record by HRJ
                  update Aug 2019 V3.5++, added col 1 * comment

      DESCRIPTION:      "This program lets you use your computer to assemble
                  code for the Motorola 6800, 6801, 6802, 6803, 6808,
                  and 68701 microprocessors.  The program is written in
                  portable C rather than BDS C.  All assembler features
                  are supported except relocation, linkage, and macros.";

      KEYWORDS:   Software Development, Assemblers, Cross-Assemblers,
                  Motorola, MC6800, MC6801;

      SEE-ALSO:   CUG113, 6800 Cross-Assembler;

      SYSTEM:           CP/M-80, CP/M-86, HP-UX, MSDOS, PCDOS, QNIX;
      COMPILERS:  Aztec C86, Aztec CII, CI-C86, Eco-C, Eco-C88, HP-UX,
                  Lattice C, Microsoft C, QNIX C;

      WARNINGS:   "This program has compiled successfully on 2 UNIX
                  compilers, 5 MSDOS compilers, and 2 CP/M compilers.
                  A port to BDS C would be extremely difficult, but see
                  volume CUG113.  A port to Toolworks C is untried."

      AUTHORS:    William C. Colley III;
*/
```

6800/6801 Cross-Assembler (Portable)

Version 3.5+

The manual such as it is.

Note to Users of Previous Versions of the Package

This version of the 6800/6801 Cross-Assembler package is a total rewrite of the old BDS C version. During the recoding, a few new "bells and whistles" found their way into the program. They are:

1) Labels can now be as long as an entire line, and all characters are significant. In older versions, only 8 characters were significant.

2) Listing control has been added to the output routine. The listing can be broken up into pages and a running header can be added to the top of each page. See the TITL and PAGE pseudo-ops for details.

3) Default extensions for the source, list, and object (hex) files are no longer supplied.

4) Include files are now supported and may be nested.

Alas, as the sage says, "There ain't no such thing as a free lunch." Massive internal changes had to be made to divorce the program from the CP/M-80 environment. These changes, the fact that full-featured, 8-bit C compilers generate less efficient code than BDS C, and the fact that I have leaned on the over-powered, slow library function printf() heavily have caused the package to run about a factor of 4 slower than the older versions. The package also takes 3-6K more disk space to store than it used to take.

On the plus side, however, the code is written in "portable" C, so all of the UNIX users, and the new crop of IBM-PC users should be able to compile and run the package almost without modification. The internal structure of the package is cleaner than ever, so it should be very easy to hack on and turn into cross-assemblers for other 8-bit processors. Finally, the source code has shrivelled almost to nothing since many tasks have been off-loaded onto standard library functions, so the need to "squeeze" the source code is gone.

Table of Contents

1.0 How to Use the Cross-Assembler Package

First, the question, "What does a cross-assembler do?" needs to be addressed as there is considerable confusion on this point. A cross-assembler is just like any other assembler except that it runs on some CPU other than the one for which it assembles code. For example, this package assembles 6801 source code into 6801 object code, but it runs on an 8080, a Z-80, an 8088, or whatever other CPU you happen to have a C compiler for. The reason that cross-assemblers are useful is that you probably already have a CPU with memory, disk drives, a text editor, an operating system, and all sorts of hard-to-build or expensive facilities on hand. A cross-assembler allows you to use these facilites to develop code for a 6801.

This program requires one input file (your 6801 source code) and zero to two output files (the listing and the object). The input file MUST be specified, or the assembler will bomb on a fatal error. The listing and object files are optional. If no listing file is specified, no listing is generated, and if no object file is specified, no object is generated. If the object file is specified, the object is written to this file in "Intel hexadecimal" format.

The command line for the cross-assembler looks like this:

```
     A68 source_file { -l list_file } { -o/-s object_file }
```

where the { } indicates that the specified item is optional. Either -o or -s is permitted but not both. -o produces an Intel hex record file; -s produces a Motorola hex record file.

Some examples are in order: file names and extensions are up to the user, these are suggested extensions.

```
a68 test68.asm                          source:   test68.asm
                                        listing:  none
                                        object:   none

a68 test68.asm -l test68.prn            source:   test68.asm
                                        listing:  test68.prn
                                        object:   none

a68 test68.asm -o test68.hex            source:   test68.asm
                                        listing:  none
                                        object:   test68.hex

a68 test68.asm -l test68.prn -s test68.s
                                        source:   test68.asm
                                        listing:  test68.prn
                                        object:   test68.s
```

The order in which the source, listing, and object files are specified does not matter. Note that no default file name extensions are supplied by the assembler as this gives rise to portability problems.

2.0 Format of Cross-Assembler Source Lines

The source file that the cross-assembler processes into a listing and an object is an ASCII text file that you can prepare with whatever editor you have at hand. The most-significant (parity) bit of each character is cleared as the character is read from disk by the cross-assembler, so editors that set this bit (such as WordStar's document mode) should not bother this program. All printing characters, the ASCII TAB character ($09), and newline character(s) are processed by the assembler. All other characters are passed through to the listing file, but are otherwise ignored.

The source file is divided into lines by newline character(s). The internal buffers of the cross-assembler will accommodate lines of up to 255 characters which should be more than ample for almost any job. If you must use longer lines, change the constant MAXLINE in file A68.H and recompile the cross-assembler. Otherwise, you will overflow the buffers, and the program will mysteriously crash.

Each source line is made up of three fields: the label field, the opcode field, and the argument field. The label field is optional, but if it is present, it must begin in column 1. A colon at the end of the label is permitted and ignored. The opcode field is optional, but if it is present, it must not begin in column 1. If both a label and an opcode are present, one or more spaces and/or TAB characters must separate the two. If the opcode requires arguments, they are placed in the argument field which is separated from the opcode field by one or more spaces and/or TAB characters.

Finally, an optional comment can be added to the end of the line or replace the entire line. Comments MUST begin with a semicolon, except Motorola's column-1 "*" comment format IS supported, Comments can follow after the last opcode or operand(s) (with a semicolon). Comments are listed but not processed. Thus, the source line looks as below, where the { } indicates that the specified item is optional; examples follow.

```
     {label}{ opcode{ arguments}}{;commentary}

  column 1
     |
     v
     GRONK   LDAA  X, OFFSET       ; This line has everything.
             STAA  MAILBOX         ; This line has no label.
     BEEP                          ; This line has no opcode.
     * This line has no label and no opcode.

     ; The previous line has nothing at all.
             END                   ; This line has no argument.
```

2.1 Labels

A label is any sequence of alphabetic or numeric characters starting with an alphabetic. The legal alphabetics are:

```
! & , . : ? [ \ ] ^ _  ` { | }  ~  A-Z  a-z
```

The numeric characters are the digits 0-9. Note that "A" is not the same as "a" in a label. This can explain mysterious U (undefined label) errors occurring when a label appears to be defined. (HRJ note: upper and lower case matters. Also, with my revision, labels ending with a ":" have the colon ignored.)

A label is permitted on any line except a line where the opcode is IF, ELSE, or ENDIF. The label is assigned the value of the assembly program counter before any of the rest of the line is processed except when the opcode is EQU, ORG, or SET.

Labels can have the same name as opcodes, but they cannot have the same name as operators or registers. The reserved (operator and register) names are:

```
A         B         AND       EQ        GE        GT
HIGH      LE        LT        LOW       MOD       NE
NOT       OR        SHL       SHR       X         XOR
```

If a label is used in an expression before it is assigned a value, the label is said to be "forward-referenced." For example:

```
L1   EQU  L2 + 1   ; L2 is forward-referenced here.
L2
L3   EQU  L2 + 1   ; L2 is not forward-referenced here.
```

2.2 Numeric Constants

Numeric constants can be formed in two ways: the Intel convention or the Motorola convention. The cross-assembler supports both.

An Intel-type numeric constant starts with a numeric character (0-9), continues with zero or more digits (0-9, A-F), and ends with an optional base designator. The base designators are H for hexadecimal, none or D for decimal, O or Q for octal, and B for binary. The hex digits a-f are converted to upper case by the assembler. Note that an Intel-type numeric constant cannot begin with A-F as it would be indistinguishable from a label. Thus, all of the following evaluate to 255 (decimal):

```
0ffH   255   255D   377O   377Q   11111111B
```

A Motorola-type numeric constant starts with a base designator and continues with a string of one or more digits. The base designators are $ for hexadecimal, none for decimal, @ for octal, and % for binary. As with Intel-type numeric constants, a-f are converted to upper case by the assembler. Thus, all of the following evaluate to 255 (decimal):

$ff   255   @377   %11111111

If a numeric constant has a value that is too large to fit into a 16-bit word, it will be truncated on the left to make it fit. Thus, for example, $123456 is truncated to $3456.

2.3 String Constants

A string constant is zero or more characters enclosed in either single quotes (' ') or double quotes (" "). Single quotes only match single quotes, and double quotes only match double quotes, as per the examples. Motorola's single-quote for char values ('?) is NOT supported, or is backslash for strings (/hello/).
In all contexts except the FCC and FDB statements,
the first character or two of the string constant are all that are used. The rest is ignored. Noting that the ASCII codes for "A" and "B" are $41 and $42, respectively, will explain the following examples:

"" and ''	evaluate to $0000
"A" and 'A'	evaluate to $0041
"AB"	evaluates to $4142

Note that the null string "" is legal and evaluates to $0000.

2.4 Expressions

An expression is made up of labels, numeric constants, and string constants glued together with arithmetic operators, logical operators, and parentheses in the usual way that algebraic expressions are made. Operators have the following fairly natural order of precedence:

Highest	anything in parentheses
	unary +, unary -
	*, /, MOD, SHL, SHR
	binary +, binary -
	LT, LE, EQ, GE, GT, NE
	NOT
	AND
	OR, XOR
Lowest	HIGH, LOW

A few notes about the various operators are in order:

1) The remainder operator MOD yields the remainder from dividing its left operand by its right operand.

2) The shifting operators SHL and SHR shift their left operand to the left or right the number of bits specified by their right operand.

3) The relational operators LT, LE, EQ, GE, GT, and NE can also be written as <, <= or =<, =, >= or =>, and <> or ><, respectively. They evaluate to $FFFF if the

statement is true, 0 otherwise.

4) The logical opeators NOT, AND, OR, and XOR do bitwise operations on their operand(s).

5) HIGH and LOW extract the high or low byte, of an expression.

6) The special symbol * can be used in place of a label or constant to represent the value of the program counter before any of the current line has been processed.

Some examples are in order at this point:

```
2 + 3 * 4                          evaluates to 14
(2 + 3) * 4                        evaluates to 20
NOT %11110000 XOR %00001010        evaluates to %00000101
HIGH $1234 SHL 1                   evaluates to $0024
@001 EQ 0                          evaluates to 0
@001 = 2 SHR 1                     evaluates to $FFFF
```

All arithmetic is unsigned with overflow from the 16-bit word ignored. Thus:

```
32768 * 2                          evaluates to 0
```

3.0 Machine Opcodes

The opcodes of the 6800 and 6801 processors are divided into groups below by the type of arguments required in the argument field of the source line. Opcodes that are peculiar to the 6801 are marked with an asterisk. A few notes on the source line syntax are in order at this point:

1) Arguments can be supplied in any order. Thus, for example, the following two source lines are equivalent:

```
LABEL     ADD A     X, 0      ;Mumble.
LABEL     ADD A     0, X      ;Mumble.
```

2) Multiple arguments may be separated from one another by spaces, tabs, or commas except that an expression must be separated from a following argument by a comma.

3) In the indexed addressing mode, the expression giving the offset may be omitted. The default offset is 0.

4) The register designators A and B may be appended to the opcode itself. Thus, for example, the following two source lines are equivalent:

```
LABEL     CLRA                ;Mumble.
LABEL     CLR       A         ;Mumble.
```

3.1 Opcodes -- No Arguments

The following opcodes allow no arguments at all in their argument fields:

ABA   ABX *   ASLD *   CBA   CLC   CLI
CLV   DAA   DES   DEX   INS   INX
LSLD *   LSRD *   MUL *   NOP   PSHX *   PULX *
RTI   RTS   SBA   SEC   SEI   SEV
SWI   TAB   TAP   TBA   TPA   TSX
TXS   WAI

3.2 Opcodes -- Register Argument or part-of-opcode:

The following opcodes EITHER accept one register argument A or B (early Motorola syntax); OR the letter A or B is part of the opcode (later assembler syntax):

ADDA
ANDA and ANDB
ASLA and ASLB
CLRA and CLRB
CMPA and CMPB
DECA and DECB
INCA and INCB
LDAA and LDAB
LSRA
ORAA and ORAB
PSHA and PSHB
PULA
SBCA and SBCB
STAA and STAB
SUBA and SUBB

3.3 Opcodes -- One Memory Argument

The opcodes in this group require one argument from the following list:

1)   X, expression where expression is 0 thru 255

2)   expression   where expression is arbitrary

The opcodes are:

JMP   JSR   STD *   STS   STX

3.4 Opcodes -- One Register or Memory Argument

The opcodes in this group require one argument as per the previous group or one of the register specifiers A or B. The opcodes are:

ASL   ASR   CLR   COM   DEC   INC
LSL   LSR   NEG   ROL   ROR   TST

3.5 Opcodes -- One Memory or Immediate Argument

The opcodes in this group require one argument from the following list:

1) X, expression where expression is 0-255

2) #expression where expression is arbitrary

3) expression where expression is arbitrary

The opcodes are:

```
ADDD *    CPX       LDD *     LDS       LDX       SUBD *
```

3.6 Opcodes -- Two Arguments

The opcodes in this group require one of the register specifiers A or B in addition to an argument from the following list:

1) X, expression where expression is 0-255

2) #expression where expression is -128 thru 255 (not permitted with opcode STA)

3) expression where expression is arbitrary

The opcodes are:

```
ADC       ADD       AND       BIT       CMP       EOR
LDA       ORA       SBC       STA       SUB
```

3.7 Opcodes -- Relative Branches

The opcodes in this group require one argument that is an expression whose value is in the range *-126 thru *+129. The opcodes are:

```
BCC       BCS       BEQ       BGE       BGT       BHI
BHS       BLE       BLO       BLS       BLT       BMI
BNE       BPL       BRA       BRN *     BSR       BVC
BVS
```

3.8 Opcodes -- Direct Addressing

Many opcodes of the 6800 and 6801 CPUs allow both one-byte direct (or zero-page) addressing and two-byte extended addressing. There is no way to explicitly call for one form of addressing over the other. The assembler will choose direct addressing if ALL of the following conditions are met:

1) The required expression contains no forward references.

2) The expression evaluates to 0-255.

3) The opcode allows direct addressing.

Otherwise, the assembler will choose extended addressing. Note that this makes it desireable to declare your zero-page RAM locations at the top of the program so that these locations will not generate forward references and foil the assembler's attempts to use direct addressing and shrink the object program.

4.0 Pseudo Opcodes

Unlike 6800/6801 opcodes, pseudo opcodes (pseudo ops) do not represent machine instructions. They are, rather, directives to the assembler. These directives require various numbers and types of arguments. They will be listed individually below.

4.1 Pseudo-ops -- CPU

By default, the assembler does not recognize the additional opcodes of the 6801 CPU. This prevents the assembler from generating invalid 6800 object code. The additional 6801 opcodes are turned on and off by this pseudo-op which requires one argument whose value is either 6800 or 6801 (decimal). Thus:

```
               CPU       6800      ;turns additional opcodes off
               CPU       6801      ;turns additional opcodes on
```

4.2 Pseudo-ops -- END

The END pseudo-op tells the assembler that the source program is over. Any further lines of the source file are ignored and not passed on to the listing. If an argument is added to the END statement, the value of the argument will be placed in the execution address line in the   hex object file. The execution address defaults to the program counter value at the point where the END was encountered. Thus, to specify that the program starts at label START, the END statement would be:

```
               END       START
```

If end-of-file is encountered on the source file before an END statement is reached, the assembler will add an END statement to the listing and flag it with a * (missing statement) error; if a hex file is requested, a start-address record will be generated.

4.3 Pseudo-ops -- EQU

The EQU pseudo-op is used to assign a specific value to a label, thus the label on this line is REQUIRED. Once the value is assigned, it cannot be reassigned by writing the label in column 1, by another EQU statement, or by a SET statement. Thus, for example, the following statement assigns the value 2 to the label TWO:

```
     TWO       EQU       1 + 1
```

The expression in the argument field must contain no forward references.

4.4 Pseudo-ops -- FCB

The FCB (Form Constant Bytes) pseudo-op allows arbitrary bytes to be spliced into the object code. Its argument is a chain of zero or more expressions that evaluate to -128 thru 255 separated by commas. If a comma occurs with no preceding expression, a $00 byte is spliced into the object code. If the expression is a string, each char is treated as a byte value. The sequence of bytes $FE $FF, $00, $01, $02 could be spliced into the code with the following statement:

```
               FCB       -2, -1, , 1, 2
```

4.5 Pseudo-ops -- FCC

The FCC (Form Constant Characters) pseudo-op allows character strings to be spliced into the object code. Its argument is a chain of zero or more string constants separated by blanks, tabs, or commas. If a comma occurs with no preceding string constant, an S (syntax) error results. The string contants are not truncated to two bytes, but are instead copied verbatim into the object code. Null strings result in no bytes of code. The message "Kaboom!!" could be spliced into the code with the following statement:

```
               FCC       "Kaboom!!"     ;This is 8 bytes of code.
```

4.6 Pseudo-ops -- FDB

The FDB (Form Double Bytes) pseudo-op allows 16-bit words to be spliced into the object code. Its argument is a chain of zero or more expressions separated by commas. If a comma occurs with no preceding expression, a word of $0000 is spliced into the code. The word is placed into memory high byte in low address, low byte in high address as per standard Motorola order. The sequence of bytes $FE $FF $00 $00 $01 $02 could be spliced into the code with the following statement:

```
               FDB       $FEFF, , $0102
```

4.7 Pseudo-ops -- IF, ELSE, ENDI

These three pseudo-ops allow the assembler to choose whether or not to assemble certain blocks of code based on the result of an expression. Code that is not assembled is passed through to the listing but otherwise ignored by the assembler. The IF pseudo-op signals the beginning of a conditionally assembled block. It requires one argument that may contain no forward references. If the value of the argument is non-zero, the block is assembled. Otherwise, the block is ignored. The ENDI pseudo-op signals the end of the conditionally assembled block. For example:

```
               IF   EXPRESSION     ;This whole thing generates
               FCB  $01, $02, $03  ;  no code whatsoever if
               ENDI                ;  EXPRESSION is zero.
```

The ELSE pseudo-op allows the assembly of either one of two blocks, but not both. The following two sequences are equivalent:

```
               IF   EXPRESSION
               ... some stuff ...
               ELSE
               ... some more stuff ...
               ENDI

     TEMP_LAB  SET  EXPRESSION
               IF   TEMP_LAB NE 0
               ... some stuff ...
               ENDI
               IF   TEMP_LAB EQ 0
               ... some more stuff ...
               ENDI
```

The pseudo-ops in this group do NOT permit labels to exist on the same line as the status of the label (ignored or not) would be ambiguous.

All IF statements (even those in ignored conditionally assembled blocks) must have corresponding ENDI statements and all ELSE and ENDI statements must have a corresponding IF statement.

IF blocks can be nested up to 16 levels deep before the assembler dies of a fatal error. This should be adequate for any conceivable job, but if you need more, change the constant IFDEPTH in file A68.H and recompile the assembler.

4.8 Pseudo-ops -- INCL

The INCL pseudo-op is used to splice the contents of another file into the current file at assembly time. The name of the file to be INCLuded is specified as a normal string constant, so the following line would splice the contents of file "const.def" into the source code stream:

```
               INCL      "const.def"
```

INCLuded files may, in turn, INCLude other files until four files are open simultaneously. This limit should be enough for any conceivable job, but if you need more, change the constant FILES in file A68.H and recompile the assembler.

4.9 Pseudo-ops -- ORG

The ORG pseudo-op is used to set the assembly program counter to a particular value. The expression that defines this value may contain no forward references. The default initial value of the assembly program counter is $0000. The following statement would change the assembly program counter to $F000:

```
               ORG       $F000
```

If a label is present on the same line as an ORG statement, it is assigned the new value of the assembly program counter.

4.10 Pseudo-ops -- PAGE

The PAGE pseudo-op always causes an immediate page ejection in the listing by inserting a form feed ('\f') character before the next line. If an argument is specified, the argument expression specifies the number of lines per page in the listing. Legal values for the expression are any number except 1 and 2. A value of 0 turns the listing pagination off. Thus, the following statement cause a page ejection and would divide the listing into 60-line pages:

```
               PAGE      60
```

4.11 Pseudo-ops -- RMB

The RMB (Reserve Memory Bytes) pseudo-op is used to reserve a block of storage for program variables, or whatever. This storage is not initialized in any way, so its value at run time will usually be random. The argument expression (which may contain no forward references) is added to the assembly program counter. The following statement would reserve 10 bytes of storage called "STORAGE":

```
     STORAGE   RMB       10
```

4.12 Pseudo-ops -- SET

The SET pseudo-op functions like the EQU pseudo-op except that the SET statement can reassign the value of a label that has already been assigned by another SET statement. Like the EQU statement, the argument expression may contain no forward references. A label defined by a SET statement cannot be redefined by writing it in column 1 or with an EQU statement. The following series of statements would set the value of label "COUNT" to 1, 2, then 3:

```
     COUNT     SET       1
     COUNT     SET       2
     COUNT     SET       3
```

4.13 Pseudo-ops -- TITL

The TITL pseudo-op sets the running title for the listing. The argument field is required and must be a string constant, though the null string ("") is legal. This title is printed after every page ejection in the listing, therefore, if page ejections have not been forced by the PAGE pseudo-op, the title will never be printed. The following statement would print the title "Random Bug Generator -- Ver 3.14159" at the top of every page of the listing:

```
               TITL      "Random Bug Generator -- Ver 3.14159"
```

5.0 Assembly Errors

When a source line contains an illegal construct, the line is flagged in the listing with a single-letter code describing the error. The meaning of each code is listed below. In addition, a count of the number of lines with errors is kept and printed on the C "stderr" device (by default, the console) after the END statement is processed. If more than one error occurs in a given line, only the first is reported. For example, the illegal label "=$#*'(" would generate the following listing line:

```
     L  0000   FF 00 00      =$#*'(     CPX       #0
```

5.1 Error * -- Illegal or Missing Statement

This error occurs when either:

1) the assembler reaches the end of the source file without seeing an END statement, or

2) an END statement is encountered in an INCLude file.

If you are "sure" that the END statement is present when the assembler thinks that it is missing, it probably is in the ignored section of an IF block. If the END statement is missing, supply it. If the END statement is in an INCLude file, delete it.

5.2 Error (-- Parenthesis Imbalance

For every left parenthesis, there must be a right parenthesis. Count them.

5.3 Error " -- Missing Quotation Mark

Strings have to begin and end with either " or '. Remember that " only matches " while ' only matches '.

5.4 Error A -- Illegal Addressing Mode

This error occurs if the index register designator X is used with a machine opcode that does not permit indexed addressing or if the immediate designator # is used with an opcode that does not permit immediate addressing.

5.5 Error B -- Branch Target Too Distant

The 6800 relative branch instructions will only reach -126 to +129 bytes from the first byte of the branch instruction. If this error occurs, the source code will have to be rearranged to shorten the distance to the branch target address or a long branch instruction that will reach anywhere (JMP or JSR) will have to be used.

5.6 Error D -- Illegal Digit

This error occurs if a digit greater than or equal to the base of a numeric constant is found. For example, a 2 in a binary number would cause a D error. Especially, watch for 8 or 9 in an octal number.

5.7 Error E -- Illegal Expression

This error occurs because of:

1) a missing expression where one is required

2) a unary operator used as a binary operator or vice-versa

3) a missing binary operator

4) a SHL or SHR count that is not 0 thru 15

5.8 Error I -- IF-ENDI Imbalance

For every IF there must be a corresponding ENDI. If this error occurs on an ELSE or ENDI statement, the corresponding IF is missing. If this error occurs on an END statement, one or more ENDI statements are missing.

5.9 Error L -- Illegal Label

This error occurs because of:

1) a non-alphabetic in column 1

2) a reserved word used as a label

3) a missing label on an EQU or SET statement

4) a label on an IF, ELSE, or ENDI statement

5.10 Error M -- Multiply Defined Label

This error occurs because of:

1) a label defined in column 1 or with the EQU statement being redefined

2) a label defined by a SET statement being redefined either in column 1 or with the EQU statement

3) the value of the label changing between assembly passes

5.11 Error O -- Illegal Opcode

The opcode field of a source line may contain only a valid machine opcode, a valid pseudo-op, or nothing at all. Anything else causes this error. Note that the unique 6801 opcodes are not valid until they are enabled with the CPU statement.

5.12 Error P -- Phasing Error

This error occurs because of:

1) a forward reference in a CPU, EQU, ORG, RMB, or SET statement

2) a label disappearing between assembly passes

5.13 Error R -- Illegal Register

This error occurs either when the register designator A or B is used with a machine opcode that does not permit it, or when the register designator is missing with a machine opcode that requires it.

5.14 Error S -- Illegal Syntax

This error means that an argument field is scrambled. Sort the mess out and reassemble.

5.15 Error T -- Too Many Arguments

This error occurs if there are more items (expressions, register designators, etc.) in the argument field than the opcode or pseudo-op requires. The assembler ignores the extra items but issues this error in case something is really mangled.

5.16 Error U -- Undefined Label

This error occurs if a label is referenced in an expression but not defined anywhere in the source program. If you are "sure" you have defined the label, note that upper and lower case letters in labels are different. Defining "LABEL" does not define "Label."

5.17 Error V -- Illegal Value

This error occurs because:

1) an index offset is not 0 thru 255, or

2) an 8-bit immediate value is not -128 thru 255, or

3) an FCB argument is not -128 thru 255, or

4) a CPU argument is not 6800 and not 6801, or

5) an INCL argument refers to a file that does not exist.

6.0 Warning Messages

Some errors that occur during the parsing of the cross-assembler command line are non-fatal. The cross-assembler flags these with a message on the C "stdout" device (by default, the console) beginning with the word "Warning." The messages are listed below:

6.1 Warning -- Illegal Option Ignored

The only options that the cross-assembler knows are -l and -o. Any other command line argument beginning with - will draw this error.

6.2 Warning -- -l Option Ignored -- No File Name
6.3 Warning -- -o Option Ignored -- No File Name

The -l and -o options require a file name to tell the assembler where to put the listing file or object file. If this file name is missing, the option is ignored.

6.4 Warning -- Extra Source File Ignored

The cross-assembler will only assemble one file at a time, so source file names after the first are ignored. To assemble a second file, invoke the assembler again. Note that under CP/M-80, the old trick of reexecuting a core image will NOT work as the initialized data areas are not reinitialized prior to the second run.

6.5 Warning -- Extra Listing File Ignored
6.6 Warning -- Extra Object File Ignored

The cross-assembler will only generate one listing and one object file per assembly run, so -l and -o options after the first are ignored.

7.0 Fatal Error Messages

Several errors that occur during the parsing of the cross-assembler command line or during the assembly run are fatal. The cross-assembler flags these with a message on the C "stdout" device (by default, the console) beginning with the words "Fatal Error." The messages are explained below:

7.1 Fatal Error -- No Source File Specified

This one is self-explanatory. The assembler does not know what to assemble.

7.2 Fatal Error -- Source File Did Not Open

The assembler could not open the source file. The most likely cause is that the source file as specified on the command line does not exist. On larger systems, there could also be priviledge violations. Rarely, a read error in the disk directory could cause this error.

7.3 Fatal Error -- Listing File Did Not Open
7.4 Fatal Error -- Object File Did Not Open

This error indicates either a defective listing or object file name or a full disk directory. Correct the file name or make more room on the disk.

7.5 Fatal Error -- Error Reading Source File

This error generally indicates a read error in the disk data space. Use your backup copy of the source file (You do have one, don't you?) to recreate the mangled file and reassemble.

7.6 Fatal Error -- Disk or Directory Full

This one is self-explanatory. Some more space must be found either by deleting files or by using a disk with more room on it.

7.7 Fatal Error -- File Stack Overflow

This error occurs if you exceed the INCLude file limit of four files open simultaneously. This limit can be increased by increasing the constant FILES in file A68.H and recompiling the cross-assembler.

7.8 Fatal Error -- If Stack Overflow

This error occurs if you exceed the nesting limit of 16 IF blocks. This limit can be increased by increasing the constant IFDEPTH in file A68.H and recompiling the cross-assembler.

7.9 Fatal Error -- Too Many Symbols

Congratulations! You have run out of memory. The space for the cross-assembler's symbol table is allocated at run-time using the C library function alloc(), so the cross-assembler will use all available memory. The only solutions to this problem are to lessen the number of labels in the source program, to use a larger memory model (MSDOS/PCDOS systems only), or to add more memory to your machine.