X68000

CROSS ASSEMBLER

X68000 Cross Assembler

User's Manual

Brian R. Anderson
2977 East 56th Avenue
Vancouver, B.C. V5S 2A2

Table of Contents

X68000 Cross Assembler -- User's Manual

Introduction

X68000 is a two pass cross assembler for the Motorola M68000 microprocessor. It accepts standard Motorola syntax (with some limitations noted below), and produces a formatted program listing file, and an object file consisting of standard Motorola S-records. The S-record format is fully explained on page 9.

System Requirements

One or more disk drives, and either CP/M-80 with 64K Bytes of memory, or MS-DOS with 128K Bytes of memory, are required to operate the assembler.

Software Supplied

The X68000 package consists of the following files:

* X68000.COM/X68000.EXE - The assembler program. Requires OPCODE.DAT to operate.

* OPCODE.DAT - A data file used to initialize the mnemonic lookup table for the assembler. This is NOT a text file. The data is stored in compact binary format. X68000 will search for this file first on the currently logged drive, then on drive A, and finally on drive B.

  With the standard configuration, there is room in the symbol table for 500 identifiers. Assembly speed is approximately 200-250 lines per minute.

Copyright

Copyright (c) 1985 by Brian R. Anderson

This software may be freely copied for personal, non-commercial use provided only that all copyright notices remain intact.

Acknowledgement

The assembler was developed in the CP/M environment using the Modula-2 compiler from Hochstrasser Computing AG (Switzerland), and later ported to the MS-DOS environment using the Modula-2 compiler from LogiTech (Palo Alto, California).

M68000 Syntax

This user's manual does NOT fully cover the M68000 syntax, but only enough of it to explain how the assembler operates. It is assumed that the reader has a copy of one of the following Motorola reference manuals (or other appropriate references):

* 16-BIT MICROPROCESSOR USER'S MANUAL
  (3rd edition) MC68000UM(AD3)

* M68000 16/32-BIT MICROPROCESSOR PROGRAMMER'S REFERENCE MANUAL
  (4th edition) M68000UM(AD4)

The standard syntax as outlined in these manuals is adhered to in most instances. However, there are a few limitations, which are explained in later sections of this user's manual.

There are three formats used for the instructions:
(The label is optional in all cases.)

Operand is implied or inherent to the instruction.

```
<label>   <opcode>
Eg.:      RTS                          ;Return from subroutine
```

Single operand.

```
<label>   <opcode>   <operand>
Eg.:      JSR        (A2)              ;A2 points to subroutine
```

Two operands. Result ends up in second operand.

```
<label>   <opcode>   <source operand>,<destination operand>
Eg.:      ADD        #4,D0             ;Add 4 to data register 0
```

Labels may consist of any combination of up to eight letters, numbers, and the underline character; labels must not start with a number. Upper and lower case letters are equivalent. This assembler does not expect a colon after the label. Some examples of valid and invalid labels follow:

```
VALID:    start  _L23  under  VAL23
INVALID:  62HOPE  error:  much_too_long
```

-Instructions Supported:

Although all Motorola mnemonics (except those specific to the 68010/68020) are supported, a few notes are needed to explain the peculiarities of X68000. Because the 68000 has such a huge instruction set, there is some overlap -- that is, there is sometimes more than one correct way to represent an operation. For example, there are three ways to add one to a data register: ADD #1,D0; ADDI #1,D0; ADDQ #1,D0. Some assemblers will substitute one form for another -- usually choosing a more efficient instruction. With this assembler, you get the mode you ask for -- period!

Motorola provides special instructions for address manipulation to ensure that the programmer is well aware when addresses are being changed. However, many 68000 assemblers will substitute ADDA (add to address) automatically even though ADD is specified. X68000 does not make this substitution, and for example, will flag ADD #4,A2 as an error; ADDA #4,A2 must be used instead.

DBRA (Decrement, Branch Always) may be used as a synonym for the less intuitive DBF (Decrement, Branch False). The DBcc looping instructions usually combine three operations: Test a condition; Decrement a register; If the register is not equal to negative one, then branch. The DBRA/DBF effectively omits the first of these steps. This is the only instruction for which a synonym exists.

The M68000 requires that all instructions be aligned to WORD boundaries. X68000 ensures this by forcing all instructions to the next highest even address (i.e., to WORD boundaries). Since all instructions require an even number of bytes, this feature would usually only make a difference if you tried to originate code (see page 5) at an odd address or if code follows directly after a data declaration reserving an odd number of bytes.

-Addressing Modes Supported:

The majority of instructions use the same 12 addressing modes (or a subset of these) to calculate the effective address of the operand(s). The Branch and Decrement/Branch group of instructions are unique in that they use only the relative addressing mode; in which case the assembler calculates the relative offset if given the absolute destination address (usually specified as a label). The special modes (CCR, SR, USP) are used in only a very few instructions.

MODE	SYNTAX
Data Register Direct:	Dn
Address Register Direct:	An
Address Register Indirect:	(An)
Address Register Indirect with Postincrement:	(An)+
Address Register Indirect with Predecrement:	-(An)
Address Register Indirect with Displacement:	d(An)
Address Register Indirect with Displacement and Index:	d(An,Xi)
Absolute Word:	xxxx.W
Absolute Long (default):	xxxxxxxx.L
Program Counter with Displacement:	d(PC)
Program Counter with Displacement and Index:	d(PC,Xi)
Relative:	<label>
Condition Code Register:	CCR
Status Register:	SR
User Stack Pointer:	USP

* n is the register number: 1..7

* SP is a synonym for A7. Eg.: MOVEM D1/D4-D6,-(SP)

* Branches and Absolute Jumps will default to their long form unless the short form is specified. This is done by applying a .S suffix to the opcode. Eg.: BRA.S <label>

* Modes requiring a displacement MUST specify same, even if the displacement is zero. Eg.: MOVE #300,0(A0,D2)

* Absolute mode will default to the long form unless word is specified. Eg.: MOVE D0,$4000.W (This allows two alternatives for specifying JMP/JSR in the word range.)
  Eg.: JSR.S <label> or JMP <label>.W (the first is preferred)

* The current value of the program counter is indicated by *. This is often used to invoke the Program Counter relative addressing mode for JMP/JSR instructions.

```
Eg.:    JMP  DEST-*-2(PC)     ;Jump to DEST using PC relative.
                              ;The '-2' is required because
                              ;the M68000 increments the PC
                              ;twice before it calculates the
                              ;new value for the PC.
```

-Assembler Directives Supported:

DIRECTIVE	SYNTAX
Originate Code or Data:	ORG
Equate a Symbol to a Value:	EQU
Force Program Counter to Word Boundary:	EVEN
Define Constant, Byte:	DC.B
Define Constant, Word (default):	DC.W
Define Constant, Long:	DC.L
Define Storage:	DS
End of Source Code:	END

* All values used in assembler directives must be defined before they are used. It is most usual that the arguments of the directives are constants.

* DC.W aligns to the next word boundary. DC.L aligns to the next long word boundary.

* DC may be used to store constant strings - maximum 10 characters per directive/line, in addition to its usual function (to reserve memory for variables). For longer strings, multiple DC directives may be used. When using this feature, DC, DC.B, DC.W, DC.L are all equivalent.
  Eg.:
```
MSG     DC.B    'This is a '  ;Maximum 10 characters per DC!
        DC.W    'very long '
        DC.L    'message.'
```

* The form <label> DC.B 10,20,30 is not supported. You must use:
```
<label>  DC.B    10
         DC.B    20
         DC.B    30
```

* DS always defines the storage in bytes, and leave the reserved memory uninitialized.
  Eg.:
```
BUFFER  DS      80              ;Reserve 80 bytes for the BUFFER.
```

* There must be at least one <CR><LF> after the END pseudo-op.

The assembler does not produce relocatable code, and therefore does not support the directives associated with relocation and/or linking. However, the 68000 instruction set makes it very easy to produce position independent code. By using no absolute mode instructions (program counter relative is available to most instructions), your resulting code can be relocated and linked without the use of a linking loader. More on this later.

-Expression Evaluation:

As mentioned above, limited string evaluation (up to ten characters) is supported for the Define Constant assembler directive. In addition, single quoted literals may be used to insert the ASCII hex value into equate statements or into instructions.

Eg.: CMPI #'0',D0 ;compares D0 with $30 ('0' = $30 in ASCII)

Limited arithmetic expression evaluation is also supported for both assembler directives and instructions. Decimal constants in the range of +/- 65535, Hexadecimal constants in the range of 0-$FFFFFFFF (32 bits), and Binary constants in the range of 0-%1111111111111111 (16 bits) are allowed. Hexadecimal and binary constants are sign-extended to 32 bits (sign extension can be defeated by including a leading zero -- Eg.: $0FF). Octal constants are not supported. The only operations that are allowed in expressions are addition and subtraction. Expressions are evaluated left-to-right, and must not contain parentheses or spaces. Although there is no limit to the number of terms in an expression, the total length of operands is limited to 19 characters. (Unsupported operations -- multiplication, division, etc. -- will be flagged as an error, usually as an Undefined Symbol). Eg.: ADDI #SIZE-4,D0

-Source Program Layout:

* Labels must appear at the left margin.

* Op-Codes must be separated from labels (or left margin, if the label is absent) by at least one tab or space. It is best to set your editor to map tabs into spaces.

* Operand(s) must be separated from op-codes by at least one tab or space.

* Operands (if two are present) must be separated by a comma.

* No spaces may appear within labels, op-codes, or operands.

* Identifiers (labels) must not start with a digit.

* All lower case characters are mapped to upper case, unless they appear within single quotes. The single quote itself must be represented by its hex value ($27), and cannot be part of a string.

* The beginning of a comment is indicated by a semicolon (anywhere on the line) or an asterisk (on the left margin only). The balance of such lines is ignored.

* See example programs starting on page 21.

Operation

The data file OPCODE.DAT must be present somewhere on the system for X68000 to operate. X68000.COM/X68000.EXE and the input file (i.e., source file) may be on any drive. The output files (Listing and S-record) will always be written to the same drive that the input file is on. A good arrangement is to have your editor along with X68000.COM/X68000.EXE and OPCODE.DAT on the A: drive, with the assembly source files on the B: drive.

Operation of the assembler is very straight forward (the hard part is creating a syntactically correct source program). Invoke the assembler by typing X68000, optionally followed by a source program filename. If you do not enter a filename, the assembler will prompt you for one.

The source program filename must adhere to normal operating system (CP/M or MS-DOS) conventions, and is usually of type extension .ASM; this will be assumed if you omit the dot (separator) and the filetype.

To assemble a file called B:TEST.ASM from the system prompt: (Operator input is underlined)

A>X68000 B:TEST<CR>

or

A>X68000<CR>

Enter Source Filename: B:TEST<CR>

After a short delay, the following messages will appear on the console screen (assuming that B:TEST.ASM had no errors):

```
                68000 Cross Assembler
          Copyright (c) 1985 by Brian R. Anderson

                Assembling B:TEST.ASM

PASS 1
PASS 2

--->  END OF ASSEMBLY
--->  0 ASSEMBLY ERROR(S).
```

The various messages will appear in sequence as the two passes are executed. Most of the error checking is done on the second pass, but some errors will be detected in both passes. Error messages will be output to the console in the following form:

```
23   LOOP   JMP    D3          ;Repeat input section
                    ^
This addressing mode not allowed here.
```

The number (23) indicates which line of the source file that the error was found on, and is followed by the line as it appeared in the file.

In most instances, an arrow (^) will point to the location of the error. Although the error detection is very good (all errors will be reported), the verbal error messages will sometimes need interpreting, especially if totally incorrect syntax is encountered.

Since X68000 has no way of knowing how tab stops have been set up on your editor and video terminal, the error arrow (^) will not be positioned correctly if tabs are included in your 68000 source code. For accurate positioning of the error arrow, you should set your editor to map tabs into spaces.

After an error, assembly will be suspended until any key on the console is struck; this allows the programmer to make note of the error before continuing. Although the listing file will contain a tally of the number of errors found, the errors will not be pointed out. The only detailed error information comes from the error messages output to the console.

The assembler will abort with an error message under three conditions: no source file; out of room in the symbol table; too many errors encountered. There is room for 500 symbols, and up to 500 errors are allowed before termination. (The operator may abort after any error by typing Control-C.)

Output File Format

The listing file contains everything that was in the source file (formatted into pages), along with the object code generated for each line of source code. Also included in the listing file is a sorted symbol table showing all identifiers defined by the programmer, along with their values. To get a print-out of the listing file, use your operating system utility (PIP, COPY, or similar) to send the file to the printer.

The S-record file may be used to upload the object code to development hardware or EPROM programmers, etc.. The format adheres to Motorola standards, as outlined below.

Three types of records are used: S0 (header), S2 (code/data), S8 (trailer). The S0 record contains a 2-byte address (always zero); the S2 and S8 records contain 3-byte addresses (S8 address is always zero, too). The S2 record is the most important, and will be explained in detail. The other records have a substantially similar format.

The S2 records provide all the address, code, and data information about the source code. Each record is autonomous in that it contains both the code or data along with the memory address where that code or data is to reside. Each record is also checked for errors by way of count and checksum fields.

Each S-record consists of up to 5 fields:

* Sn - Indicates the type of field (where n is 0 - 9).
* Count - Byte count; includes address bytes and checksum, but excludes S-type & count.
* Address - Starting address of this line of code or data.
* Code/Data - 16 bytes of program code or data. Contains source filename for S0 record; absent for S8 record.
* Checksum - Least significant byte of 1's complement of the sum of count, address, and code/data bytes.

Manual Linking

Linking consists of joining two or more separately assembled modules together to form one executable program. To make manual linking practical, it is important to make all modules position independent. That means that they can be loaded at any address in memory, and they will still execute properly. In practical terms, that means you must avoid the use of absolute address references in your code modules.

Although absolute address references prevent complete position independence, they are impossible to avoid altogether. A program may be made of several modules, each with one or more subroutines. When the main program, or any subroutine, calls a subroutine that is outside its own module (a very common situation!), it must know the absolute address of that subroutine. In the case of global data (data that is accessible by all parts of the program), absolute addresses are also needed. (However, all local references can easily be made position independent through the use of relative address instructions or a stack frame. See figure 10 (page 22) for an example of relative addressing.)

An assembler and linker often team up to provide these cross references to all parts of a large program. The programmer specifies all objects (data and subroutines) that are to be accessed from outside a module, as GLOBAL. At assembly time, additional information is included with the assembled object file. That information includes the name of all GLOBAL objects, and their relative position within that module. At link time, the linker collects this information from all modules, and resolves accesses to GLOBAL information. The code produced by X68000 DOES NOT include relocation information, so linking must be done manually.

-Tracking Global Data:

To manually perform linking, the first step is to choose one area in memory where all global data is to be located. Global data should be kept to a minimum anyway, so keeping track of this area should be no real burden. Each time you need a new global data item, write down its name, size, and relative position (in bytes) within the global data area. When referring to these absolute addresses, simply use the starting address plus the relative position.

```
Eg.:
GLODAT    EQU       $0200           ;Global data starts here
*
*
          MOVE      D0,GLODAT+4     ;Store at 2nd Long Word
                                    ;of Global Data area
*
```

-Tracking Subroutines:

Manually tracking subroutines is a little more difficult, as it involves some calculation of addresses, and some reassembly of modules. The first assembly is done with all modules (except possibly the main program module) originated at zero, and with equate statements for each external global subroutine name showing the address of that subroutine as zero. The assembler will calculate the starting address of each internal subroutine (in fact, of each instruction), as well as the total length of all modules. Record all of this information.

Now, you have a few calculations to do: figure out the total memory requirements, and determine the new starting address for each module by stacking them end to end (this is called a Link Map). Based on the actual starting address of each module and the relative address of subroutines within that module, calculate the absolute address of each subroutine. Next, change all EQU statements that refer to global subroutines to their real addresses; and change all ORG statements to reflect their position within the stack of modules that you have created. Finally, reassemble all modules.

Linking is now just a matter of concatenating all of the S-record files (with PIP or COPY) and then eliminating extraneous S0 and S8 records (leaving only the first S0 record, the S2 records from all modules, and the last S8 record). The S-record file is then ready to upload to a PROM programmer. All sections will automatically be loaded to their correct areas in memory due to the nature of S-records (see page 9 for more details on S-records).

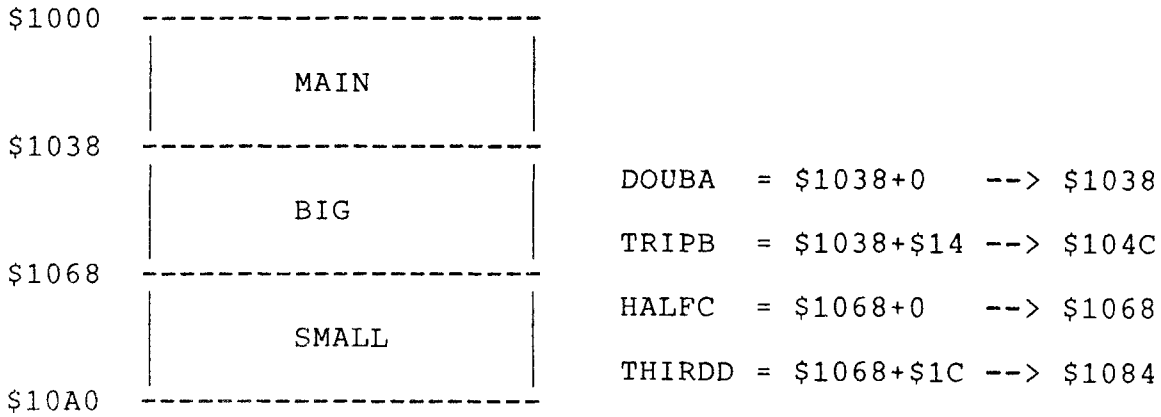
The following simple example illustrates manual linking of three modules. The first module consists of a main program, while the others each consist of 2 subroutines. Four items of global data are shared among the modules.

The results of the initial assembly are shown in figures 1 through 3 (pp 13-15). The main module declares four data items. Notice that their offsets reflect the size of the data (one is LONG (4 bytes), while the others are WORD (2 bytes)). It is up to the programmer to calculate these offsets.

Information derived from the first assembly listings:

* MAIN ends at $1038. (BIG will start here)
* DOUBA subroutine starts at beginning of BIG.
* TRIPB subroutine starts $14 bytes beyond the beginning of BIG.
* BIG ends at $30. (SMALL will start at $1038+$30 --> $1068)
* HALFC subroutine starts at beginning of SMALL.
* THIRDD subroutine starts $1C bytes beyond the beginning of SMALL.
* SMALL ends at $38. (This information is not needed, because SMALL is the last module)

Proposed Link Map:
(BIG and SMALL stacked on top of MAIN.)



The addresses of the subroutines must now be substituted for the zeros in the EQU statements that refer to them in MAIN. Also, the new ORG values indicating the start of modules BIG and SMALL must be substituted for the zeros initially used. Figures 4 through 6 (pp 16-18) show the final listings with all external address references resolved.

Figure 7 (page 19) shows the concatenated S-records, while figure 8 (page 19) shows the same file with the extra S0/S8 records removed. The S-record in figure 8 is now ready for uploading.

```
                        ************************************************
                        *
                        *         Main Program
                        *             of
                        *       Manual Link Example
                        *
                        ************************************************

                                ORG     $100            :Global Data Area
000100  00000000        A       DC.L    0               :Offset = 0
000104  0000            B       DC.W    0               :Offset = 4
000106  0000            C       DC.W    0               :Offset = 6
000108  0000            D       DC.W    0               :Offset = 8
        00000000        DOUBA   EQU     $0000           :External Subroutine
        00000000        TRIPB   EQU     $0000           :External Subroutine
        00000000        HALFC   EQU     $0000           :External Subroutine
        00000000        THIRDD  EQU     $0000           :External Subroutine

                                ORG     $1000
001000  203C00000100    MAIN    MOVE.L  #$100,D0
                        *
001006  23C000000100            MOVE.L  D0,A
00100C  4EB900000000            JSR     DOUBA           :A doubled
                        *
001012  33C000000104            MOVE    D0,B
001018  4EB900000000            JSR     TRIPB           :B trippled
                        *
00101E  33C000000106            MOVE    D0,C
001024  4EB900000000            JSR     HALFC           ;C halved
                        *
00102A  33C000000108            MOVE    D0,D
001030  4EB900000000            JSR     THIRDD          :D Divided by 3
                        *
001036  4E40                    TRAP    #0              :Return to DOS
001038                          END


--->  END OF ASSEMBLY
--->  0 ASSEMBLY ERROR(S).

            * * *  Symbolic Reference Table  * * *


A       : 00000100  ;  B       : 00000104  ;  C       : 0000010é
D       : 00000108  ;  DOUBA   : 00000000  ;  HALFC   : 00000000
MAIN    : 00001000  ;  THIRDD  : 00000000  ;  TRIPB   : 00000000
```

Figure 1

Main program module. Global data declared, and offsets calculated. Since addresses of external subroutines are unknown, zeros are inserted in the equate statements that will eventually contain their absolute addresses.

```
                        ***********************************************
                        *       Module used by Main
                        *       No overflow checking in subroutines
                        ***********************************************

        00000100        GLOBDAT EQU     $100            ;Start of Data

                                ORG     0
000000  3F00            DOUBA   MOVE    D0,-(SP)        ;Push register
000002  203900000100            MOVE.L  GLOBDAT+0,D0    ;Get Global Data
000008  D080                    ADD.L   D0,D0           ;Double it
00000A  23C000000100            MOVE.L  D0,GLOBDAT+0    ;Put it back
000010  301F                    MOVE    (SP)+,D0        ;Restore register
000012  4E75                    RTS
                        *
000014  48A7C000        TRIPB   MOVEM   D0/D1,-(SP)     ;Save registers
000018  303900000104            MOVE    GLOBDAT+4,D0    ;Get Global Data
00001E  3200                    MOVE    D0,D1           ;Make a copy
000020  D040                    ADD     D0,D0           ;Double it
000022  D041                    ADD     D1,D0           ;Triple
000024  33C000000104            MOVE    D0,GLOBDAT+4    ;Put it back
00002A  4C9F0003                MOVEM   (SP)+,D0/D1     ;Restore registers
00002E  4E75                    RTS
                        *
000030                          END


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

            * * *  Symbolic Reference Table  * * *


DOUBA    : 00000000  :  GLOBDAT   : 00000100  :  TRIPB    : 00000014
```

Figure 2

BIG module containing DOUBA and TRIPB subroutines. Start of Global Data Area is specified with EQU statement. All global data is offset from this.

```
                        ********************************************
                        *       Subroutines for Main
                        ********************************************

        00000100        GLOBDAT EQU     $100

                                ORG     0
000000  2F00            HALFC   MOVE.L  D0,-(SP)        :Push
000002  303900000106            MOVE.W  GLOBDAT+6.D0    ;Get Global Data
000008  80FC0002                DIVU    #2,D0           :Half
00000C  0280FFFFFFFF            ANDI.L  #$FFFF,D0       ;Dump remainder
000012  33C000000106            MOVE.W  D0,GLOBDAT+6    :Put back data
000018  201F                    MOVE.L  (SP)+.D0        ;Pull
00001A  4E75                    RTS
                        *
00001C  2F00            THIRDD  MOVE.L  D0,-(SP)        :Push
00001E  303900000108            MOVE.W  GLOBDAT+8.D0    ;Get data
000024  80FC0003                DIVU    #3,D0           :Third
000028  0280FFFFFFFF            ANDI.L  #$FFFF,D0       ;Dump remainder
00002E  33C000000108            MOVE.W  D0,GLOBDAT+8    :Put back data
000034  201F                    MOVE.L  (SP)+.D0        ;Pull
000036  4E75                    RTS
                        *
000038                          END


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

                * * *  Symbolic Reference Table  * * *


GLOBDAT   : 00000100  :  HALFC    : 00000000  :  THIRDD    : 0000001C
```

Figure 3

SMALL module with HALFC and THIRDD subroutines.

```
                        ********************************************************
                        *
                        *         Main Prooram
                        *             of
                        *       Manual Link Example
                        *
                        ********************************************************

                                ORG     $100            :Global Data Area
000100  00000000        A       DC.L    0               :Offset = 0
000104  0000            B       DC.W    0               :Offset = 4
000106  0000            C       DC.W    0               :Offset = 6
000108  0000            D       DC.W    0               :Offset = 8
        00001038        DOUBA   EQU     $1038           :External Subroutine
        0000104C        TRIPB   EQU     $104C           :External Subroutine
        00001068        HALFC   EQU     $1068           :External Subroutine
        00001084        THIRDD  EQU     $1084           :External Subroutine

                                ORG     $1000
001000  203C00000100    MAIN    MOVE.L  #$100,D0
                        *
001006  23C000000100            MOVE.L  D0,A
00100C  4EB900001038            JSR     DOUBA           :A doubled
                        *
001012  33C000000104            MOVE    D0,B
001018  4EB90000104C            JSR     TRIPB           :B trippled
                        *
00101E  33C000000106            MOVE    D0,C
001024  4EB900001068            JSR     HALFC           :C halved
                        *
00102A  33C000000108            MOVE    D0,D
001030  4EB900001084            JSR     THIRDD          :D Divided by 3
                        *
001036  4E40                    TRAP    #0              :Return to DOS
001038                          END


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

            * * *  Symbolic Reference Table  * * *


A       : 00000100  :  B       : 00000104  :  C       : 00000106
D       : 00000108  :  DOUBA   : 00001038  :  HALFC   : 00001068
MAIN    : 00001000  :  THIRDD  : 00001084  :  TRIPB   : 0000104C
```

Figure 4

MAIN module must be altered and reassembled, due to external subroutine references. Addresses calculated earlier (see page 12), have been inserted into the EQU references.

```
                        ************************************************
                        *       Module used by Main
                        *       No overflow checking in subroutines
                        ************************************************

        00000100        GLOBDAT EQU     $100            ;Start of Data

                                ORG     $1038
001038  3F00            DOUBA   MOVE    D0,-(SP)        ;Push register
00103A  203900000100            MOVE.L  GLOBDAT+0,D0    ;Get Global Data
001040  D080                    ADD.L   D0,D0           ;Double it
001042  23C000000100            MOVE.L  D0,GLOBDAT+0    ;Put it back
001048  301F                    MOVE    (SP)+,D0        ;Restore register
00104A  4E75                    RTS
                        *
00104C  48A7C000        TRIPB   MOVEM   D0/D1,-(SP)     ;Save registers
001050  303900000104            MOVE    GLOBDAT+4,D0    ;Get Global Data
001056  3200                    MOVE    D0,D1           ;Make a copy
001058  D040                    ADD     D0,D0           ;Double it
00105A  D041                    ADD     D1,D0           ;Triple
00105C  33C000000104            MOVE    D0,GLOBDAT+4    ;Put it back
001062  4C9F0003                MOVEM   (SP)+,D0/D1     ;Restore registers
001066  4E75                    RTS
                        *
001068                          END


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

              * * *  Symbolic Reference Table  * * *


DOUBA     : 00001038  :  GLOBDAT   : 00000100  :  TRIPB     : 0000104C
```

Figure 5

BIG module with ORG changed to coincide with the Link Map.

```
                        ********************************************
                        *       Subroutines for Main
                        ********************************************

        00000100        GLOBDAT EQU     $100

                                ORG     $1068
001068  2F00            HALFC   MOVE.L  D0,-(SP)        ;Push
00106A  303900000106            MOVE.W  GLOBDAT+6,D0    ;Get Global Data
001070  80FC0002                DIVU    #2,D0           ;Half
001074  0280FFFFFFFF            ANDI.L  #$FFFF,D0       ;Dump remainder
00107A  33C000000106            MOVE.W  D0,GLOBDAT+6    ;Put back data
001080  201F                    MOVE.L  (SP)+,D0        ;Pull
001082  4E75                    RTS
                        *
001084  2F00            THIRDD  MOVE.L  D0,-(SP)        ;Push
001086  303900000108            MOVE.W  GLOBDAT+8,D0    ;Get data
00108C  80FC0003                DIVU    #3,D0           ;Third
001090  0280FFFFFFFF            ANDI.L  #$FFFF,D0       ;Dump remainder
001096  33C000000108            MOVE.W  D0,GLOBDAT+8    ;Put back data
00109C  201F                    MOVE.L  (SP)+,D0        ;Pull
00109E  4E75                    RTS
                        *
0010A0                          END


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

                * * *  Symbolic Reference Table  * * *


GLOBDAT   : 00000100  ;  HALFC    : 00001068  ;  THIRDD    : 00001084
```

Figure 6

SMALL module with ORG changed to coincide with the Link Map.

```
S00B00004D41494E2E41534DC0
S20E0001000000000000000000000000F0
S2140010002030000010023C0000001004EB9000093
S2140010101038330000001044EB90000104C33C035
S2140010200000001064EB90000106833C00000010839
S20C0010304EB9000010B44E408A
S804000000FC

S00A00004249472E41534D14
S20C0010383F0020390000010012
S2140010400D08023C000000100301F4E7548A7C000A6
S2140010503039000001043200D040D04133C00000D7
S20C00106001044C9F00034E75CD
S804000000FC

S00C0000534D414C4C2E41534D6B
S20C0010682F0030390000106DC
S2140010708FC00020280FFFFFFFF33C00000010675
S214001080201F4E752F00303900000108B0FC000339
S21400109002B0FFFFFFFF33C000000108201F4E75CF
S804000000FC
```

Figure 7

Concatenated S-record file before extra S-records are removed.

```
S00B00004D41494E2E41534DC0
S20E0001000000000000000000000000F0
S2140010002030000010023C0000001004EB9000093
S2140010101038330000001044EB90000104C33C035
S2140010200000001064EB90000106833C00000010839
S20C0010304EB9000010B44E408A
S20C0010383F0020390000010012
S2140010400D08023C000000100301F4E7548A7C000A6
S2140010503039000001043200D040D04133C00000D7
S20C00106001044C9F00034E75CD
S20C0010682F0030390000106DC
S2140010708FC00020280FFFFFFFF33C00000010675
S214001080201F4E752F00303900000108B0FC000339
S21400109002B0FFFFFFFF33C000000108201F4E75CF
S804000000FC
```

Figure 8

Final S-record files. Ready for uploading to a PROM programmer or evaluation circuit.

Although this is a simplistic example, the same method can be used to manually link virtually any group of modules. The main points to remember:

* Avoid absolute references. Use relative branches for all internal jumps. Use PC relative addressing (or a stack frame) for access to local variables. Absolute references will be required only for access to global data and external subroutines.

* Keep all global data together, and calculate the offset of each based on their size and position. The correct place to declare global data is in the main program module.

* ORG the main program at whatever location you wish.

* Provide EQU statements to specify the starting address of all external subroutines. These are initially set to zero.

* ORG all modules (with the possible exception of main) at zero.

* Assemble all modules.

* Create a Link Map by stacking all modules on top of the main program.

* Calculate the starting addresses of all subroutines based on their position in the Link Map, and their relative offset from the beginning of their own module. The necessary information regarding these addresses is obtained from the first assembly listings.

* Substitute the absolute address of all external subroutines (calculated with the aid of the Link Map) in the EQU statements specifying their addresses.

* Change the ORG statements of all modules (except the main program), based on the Link Map that you have created.

* Reassemble all modules.

* Concatenate S-records and eliminate interior S0/S8 records.

```
                        ********************************************
                        *
                        *       flags: A0       ;pointer to array
                        *       iter:  D0
                        *       count: D1
                        *       i:     D2
                        *       prime: D3
                        *       k:     D4
                        *
                        ********************************************

                                org     $1000
        00001FFF        size    equ     8191            ;array size
001000  00              flags   ds      size            ;array of boolean
        0000000A        iter    equ     10              ;10 iterations
        00000001        TRUE    equ     %00000001       ;
        00000000        FALSE   equ     %00000000       ;
        000000E4        tutor   equ     228             ;return to OS

                                org     $3000
003000  303C000A        start   move    #iter,d0        ;set max iterations
003004  207C00001000            movea.l #flags,a0       ;point to flags
00300A  7200            again   moveq   #0,d1           ;count <-- 0
00300C  7400                    moveq   #0,d2           ;i <-- 0
00300E  11BC00012000    setflg  move.b  #TRUE,0(a0,d2)  ;flags[i] <-- TRUE
003014  5242                    addq    #1,d2           ;i <-- i + 1
003016  B47C1FFF                cmp     #size,d2        ;i > size?
00301A  6FF2                    ble.s   setflg          ;
00301C  7400                    moveq   #0,d2           ;i <-- 0
00301E  0C3000012000    next    cmpi.b  #TRUE,0(a0,d2)  ;flags[i] = TRUE?
003024  661C                    bne.s   notrue          ;
003026  3602                    move    d2,d3           ;prime <-- i + i + 3
003028  D642                    add     d2,d3           ;
00302A  5643                    addq    #3,d3           ;
00302C  3802                    move    d2,d4           ;k <-- i + prime
00302E  D843                    add     d3,d4           ;
003030  B87C1FFF        more    cmp     #size,d4        ;k <= size?
003034  6E0A                    bgt.s   nxtcnt          ;
003036  11BC00004000            move.b  #FALSE,0(a0,d4) ;flags[k] <-- FALSE
00303C  D843                    add     d3,d4           ;k <-- k + prime
00303E  60F0                    bra.s   more            ;
003040  5241            nxtcnt  addq    #1,d1           ;count <-- count + 1
003042  5242            notrue  addq    #1,d2           ;i <-- i + 1
003044  B47C1FFF                cmp     #size,d2        ;i > size
003048  6FD4                    ble.s   next            ;
00304A  5340                    subq    #1,d0           ;iter <-- iter - 1
00304C  6600FFBC                bne     again           ;last iteration?
003050  3E3C00E4                move    #tutor,D7       ;escape to supervisor
003054  4E4E                    trap    #14             ;
003056                          end


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).
               ! ! !  Symbolic Reference Table  ! ! !

AGAIN     : 0000300A  !  FALSE     : 00000000  !  FLAGS     : 00001000
ITER      : 0000000A  !  MORE      : 00003030  !  NEXT      : 0000301E
NOTRUE    : 00003042  !  NXTCNT    : 00003040  !  SETFLG    : 0000300E
SIZE      : 00001FFF  !  START     : 00003000  !  TRUE      : 00000001
TUTOR     : 000000E4  !
```

Figure 9

Sieve of Eratosthenes
As implemented for Motorola Educational Board Computer MEX68KECB/D2.

```
                        ************************************************
                        *
                        *  B.C.D to 7-Segment Conversion
                        *
                        *
                        *     Entry D0 = BCD digit
                        *
                        *     Exit  7-Segment equivalent --> D0
                        *
                        ************************************************

                                org     0

000000  2F08            dyscod  move.l  a0,-(SP)          ;safekeeping
000002  02800000000F            andi.l  #$000F,d0         ;clean BCD
000008  103B0006                move.b  look-2-*(PC,d0),d0 ;get 7-Segment code
00000C  205F                    movea.l (SP)+,a0          ;restore register
00000E  4E75                    rts

                        *
                        *       Look-up Table
                        *
000010  3F              look    dc.b    $3F               ; '0'
000011  06                      dc.b    $06               ; '1'
000012  5B                      dc.b    $5B               ; '2'
000013  4F                      dc.b    $4F               ; '3'
000014  66                      dc.b    $66               ; '4'
000015  6D                      dc.b    $6D               ; '5'
000016  7D                      dc.b    $7D               ; '6'
000017  07                      dc.b    $07               ; '7'
000018  7F                      dc.b    $7F               ; '8'
000019  67                      dc.b    $67               ; '9'

00001A                          end


--->   END OF ASSEMBLY
--->   0 ASSEMBLY ERROR(S).

            * * *  Symbolic Reference Table  * * *


DYSCOD    : 00000000  :  LOOK     : 00000010  :
```

Figure 10

B.C.D. to 7-Segment Display Driver
Demonstrates the use of PC relative addressing
to access Local data in position independent code.