Motorola Free Ware Cross Assemblers
release TER_2.0

1.0 COPY RIGHT NOTICE

You may not distribute these cross assemblers and charge more than a nominal fee for the distribution media itself. This software is the property of Motorola and is made available as free ware for the purpose of developing software for their microprocessors only. "Free Ware" means that no charge is made for copying or for the use of computer software but that all other rights including but not limited to use of the software in whole or in part in other programs remains with Motorola. Software means the source code and all versions of object code or computer coded instructions in whole or in part. All copies must carry this notice.

1.1 Disclaimer

The cross assemblers are supplied "as is" without warranty of any kind expressed or implied. They are believed to be accurate but the user takes upon him or herself the entire responsibility and consequences of their use. Neither Motorola nor any of the contributing authors warrant that the software is without errors, will operate without interruption or will be compatible with any software or hardware possessed or to be possessed by the user or that the use of the software or any of its parts will result in any economic advantage or reduction in cost. Neither Motorola nor any of the contributing authors will be liable for any special or incidental or consequential damages, even if informed of the possibility of such damages in advance.

1.2 Contributing Authors

Original Code E.J. Rupp 12/11/84

Releases beginning with TER Terry E. Rogers
(modifications & manual only) 6255 Henryk Woods Rd.
Clay, New York 13041

Note: This software is a direct descendant of the E.J. Rupp version. Other versions may exist.

2.0 INDEX

_

3.0 GENERAL

3.1 Assemblers

These assemblers are updated versions of the original, two pass Motorola 8 bit uC/uP free ware cross assemblers. The assemblers are named as* where '*' is any of 0,1,4,5,9 or 11. On MS-DOS systems, the assemblers have the extension .EXE. Other systems do not have extensions.

Example:	AS11	Amiga or MAC 68HC11 cross assembler
	AS11.EXE	MS-DOS 68HC11 cross assembler

Amiga versions have companion *. info files where * is here as5, as11 etc. The .info files are for WorkBench visibility. Amiga versions may also include scripts and companion *. info files titled "Assemble 68HC11" e.g. which endow the assemblers with extended WorkBench selection of source code files. This manual, updates and other documentation have the extension *.doc on all systems. *. doc files are linked to system ASCII readers at GUI (graphic user interface) level on the MAC and Amiga. (WorkBench is the Amiga GUI super application program analogous to MAC DeskTop.) See section 8 for host OS specifics.

The file Update.doc (if any) contains enhancement descriptions.

3.2 Command Line Format and Options

The following pertains to MS-DOS and Amiga systems. The MAC does not have a command line user interface.

The command line looks like this :

```
as* file1 [file2] ... [ - option1 option2 ...]
```

Where:

- as* is one of the assemblers mentioned above
- file<n> is an assembler source code file name complete with path (if not in current directory) in the host DOS format
- option<n> is a UNIX like assembler option or command

file1, file2, file<n> are command line files as opposed to include files (there are subtle differences). One or more spaces separates as* from file1, file1 from file2 etc. and file<n> from the minus sign preceding option1 and option 1 from option2 etc. At least one assembler source code file is required. Options are, of course, not required.

file<n> may have the extension *. asm on all systems. Any other extension acceptable to the host DOS is also permitted. The file<n> files are assembled together with file1[. asm] as if they were all one file (no linking is performed and all symbols are global) to make file1.S19, a Motorola S record output in the default directory. Source code files are left intact.

file<n> must be in "flat ASCII", i.e. must contain only alphanumeric, tab and end of line characters and must not contain formatting characters other than tab. Many word processors will not produce a flat ASCII file even using a "text only" store or ASCII conversion option. Form feeds and ANSI console control characters will probably be present. Acceptable editors available on many systems include EMACS, MEMACS, VEDIT, vii (UNIX), ED (UNIX & AmigaDOS), EDIT (AmigaDOS), EDLIN (MS-DOS) and the integrated editor in many MS-DOS, MAC and Amiga languages such as C, Compiled BASIC, FORTRAN etc. There are many others. This is not a recommendation for any.

Example:

```
as5  BlackBox Serial -l  cre
```

This command assembles files " BlackBox" and "Serial" using the 6805 cross assembler with an output listing and a cross reference table producing S record "BlackBox.S19" or "BLACKBOX.S19".
The assemblers accept options from the command line. These options are the following:

```
l         enable output listing.
nol       disable output listing (default).
cre       generate cross reference table.
s         generate a symbol table.
c         enable cycle count.
noc       disable cycle count.
crlf      enable <CR,LF> (non MS-DOS systems)
nnf       number include files contiguous with command line files
p50    page break approximately every 50 lines
```

Command line options may not be combined as they may sometimes be in UNIX. There is only one minus sign.

Listings, tables, error messages and warning messages are all directed to std.out (standard output) which may vary between systems but is usually the CRT console. Printed or file output typically must use "indirection" as defined on that system. See section 8.

3.3 Compatibility with Previous Versions

The cross assemblers will accept source code files written for the original cross assembler porting to the "IBM PC" (MS-DOS systems) on 4/13/87 providing that the files are supplied as command line arguments as shown above. An "end" statement must be supplied as the last executable pseudo-op to be used as an include file. A file so modified may be used with either the original or this version provided no further changes are made. ("End" is an ignored pseudo-op in the original.)

Source code written using the added features in this version are not compatible with the original.

3.4 Compatibility between Host DOS Systems and Support Equipment

MAC and Amiga versions of these cross assemblers will accept source files originating on MS-DOS machines. MS-DOS versions will not accept source code originating on MAC or Amigas without modification. The end of line character on MS-DOS machines is <CR,LF> or $0D$0A. The end of line character on the Amiga and MAC is <LF> or $0A only. The MAC and Amiga versions search for both MS-DOS and their own EOL character. You may use the search and replace features of the editors mentioned above (except the embedded language editors) to replace EOLs for cross system compatibility. Consult your manual for search and replace and "quote" character.

Some PROM burners, simulators and ROM manufacturers will not accept S1 records with a <LF> EOL character. Use option - crlf to change the S record output only to MS-DOS <CR,LF> EOL format if using the MAC or Amiga. Other output remains in the host (D)OS format.

4.0 ASSEMBLER SOURCE CODE FORMAT

The general source code format is:

```
[<label>][:][...< Opcode>][...<Operand>] [... (; | *)][<comment>]
   or
[; | *] <comment>
   or
<...>
     (an  Opcode is the Mnemonic)
where:
   < a  > means 'a' is a user supplied item
   [  ] means optional item
   a | b means either item a OR b is acceptable
   ...  means blanks or blank line (actually any white space '     '|
                                                    '\t' | '\n')
```

A label is optionally followed by a colon. A label may appear on a line by itself. Labels are not required. White space is a required delimiter where shown; even preceding comments. White space is a space, tab or new line character. Empty lines are treated as comments. A line beginning with '*' or ';' in the first column is also a comment line. A line having ';' as the first character, even if not in the first column, is a comment line.

The operand is, of course, optional if the Opcode or Pseudo- Opcode does not require it. Comments are preceded by ';' or '*" or just white space if it can be unambiguously understood that what follows must be a comment. '*' is handled in the same manner as in the previous cross assembler; a search is made for '*' only in the first column on a line. If you make a mistake and leave out the Operand when it's required or use '*' to introduce a comment after a label, the rest of the line will be picked up as the (Pseudo) Opcode or Operand and produce an error. However, ';' will introduce a comment on any line at any point provided that white space is immediately to the left. That is, ';' is not a delimiter for Opcodes etc. Generally it's best to use ';' to introduce comments so as to avoid errors and confusion with '*' when used in an expression (shown below).

4.1 Labels & Symbols

Symbol:

A string of characters with a non-initial digit. The string of characters may be from the set:

```
[a-z][A-Z]_.[0-9]$@
```

(. and _ count as non-digits). The '$' and '@' count as a digit to avoid confusion with hexadecimal and octal constants. All characters of a symbol are significant, with upper and lower case characters being distinct. The maximum number of characters in a symbol is currently set at 15. Symbols have a integer value which may be zero.

A symbol ending in the character '@' is redefinable. That is, its value, either an address or EQU value, may be altered without causing a redefined symbol error. Note that in the cross reference and symbol table print outs that only the most recent value will be printed.

A symbol is defined from the point in the file where it first occurs as a label onward. Using a symbol as an Operand (forward reference) does not define it. A symbol is logically "undefined" (but still may have a value) at the beginning of each pass.

Label:

A symbol starting in the first column is a label and may optionally be ended with a ':'. A label may appear on a line by itself and is then interpreted as:

```
       Label: EQU *
```

(That is the value of the symbol is set to the value of the program counter. See section 5.2.)

A label may also be a redefinable symbol. This is an alternative to local symbols when working with large files if the label is "very local" so that the programmer is able to be certain of the label value (address) when assembled. Redefinable symbols should not be used as forward references i.e. should not occur as an operand as in "BNE LOOP@" before LOOP@ is defined. The following is correct.

```
           ldaa #$ff
        loop@             ;busy loop #1
           deca
           bne loop@      ;branches to busy loop #1
           ldaa #$ff
        loop@             ;busy loop #2
           deca
           bne loop@      ;branches to busy loop #2
```

The following will not cause an assembler error but will not do what you expect. Redefinable symbols are not checked for phasing errors.

```
           cmpa #$3E
           beq loop@      ;branching into the unknown (actually a phasing
error)
           ldaa #0
        loop@:   sba #$10  ;the branch will NOT go here
           ldaa #$FF
        loop@:
           deca           ;it may go here instead (or even below this point)
           bne loop@
```

Check for this type of error if you get a branch out of range. Another possible error is caused by conditional assembly and include files. Do not nest INCLUDE <file> statements within loops using redefinable variables (see section 5.4). They could branch into a redefinable with the same name inside the include file. For example:

```
        loop@             ;might not branch here
           deca
           ifnd Data
          include  < data.i>
        ;might branch in here instead
           endif
           beq  loop@     ;branching who knows where
```

The problem is especially vexing if symbol Data is sometimes defined and not during other assemblies.

4.2 Opcode (Mnemonic)

Mnemonic (Opcode): A symbol preceded by at least one whitespace character. Upper case characters in this field are converted to lower case before being checked as a legal mnemonic. Thus `  nop', `NOP' and even `NoP' are recognized as the same mnemonic.

Note that register names that sometimes appear at the end of a mnemonic (e.g. nega or stu) must not be separated by any whitespace characters. Thus `clra' means clear accumulator A, but that ` clr a' means clear memory location `a'.

Mnemonics recognized are those listed in Motorola documentation for a particular processor or micro-computer. See section 6.x for the machine being used.

4.3 Operand

Operand: Follows mnemonic, separated by at least one whitespace character. The contents of the operand field(s) is interpreted by each instruction. In general, it may be a constant, symbol or an expression. The assemblers use all after Opcode and before ';' as a possible Operand(s).

4.3.1 Constants

Constants are constructed as:

```
'       followed by ASCII character
$ followed by hexadecimal constant
@ followed by octal constant
% followed by binary constant
digit decimal constant
```

note: ASCII constant values are those of the host DOS or OS and may vary if not the alphanumeric standards.

The following are legal constants.

```
'A          has value $41
'a          has value $61
$F2         is 242 decimal
$12EA       is 4842 decimal
@73         is 59 decimal
%10110      is $16
255         is $FF

Example:     ldaa  #$FF
```

note: The cross assemblers do not check the validity of a constant and will attempt to interpret errors with unpredictable results. 2F00 = 2 for example. The assemblers usually will flag a constant too large for the operation. However, the cross assembler itself may crash in a manner dependent upon the host (D)OS if a constant is too large for the host CPU type "int" in the C language.

4.3.2 Symbols

If the Operand is a symbol, the symbol value is used as the operand. The symbol may be an address (label) or data value. For example:

```
Max_Char = 10     ;input buffer length = value 10
ROM EQU $E000     ;ROM address start = $E000
   ORG 0          ;RAM
InBuf:            ;input buffer
   RMB  Max_Char   ;reserve Max_Char=10 bytes
...
   ORG ROM        ;change PC to ROM=$E000
...
```

The symbol may be redefinable in which case the current symbol value is used. One example, "local" branch labels, was given in section 4.2. You may also use redefinable symbols as operands for other operations such as immediate data; again as an alternative to local symbols. You should use some restraint and avoid nested include files between data definition and use as described in section 4.2.

4.3.3 Expressions

Expressions may consist of symbols, constants or the character '*' (denoting the current value of the program counter) joined together by one of the operators: +-*/%&|^. The operators are the same as in C. The entire expression is treated as one Operand field so no white space is permitted in the expression.

Expression Operators

```
+  add
-  subtract
*  multiply
/  divide
%  remainder after division
&   bitwise and
|   bitwise or
^   bitwise exclusive-or
```

Special Expression Symbol

```
*  current value of PC
```

Expressions are evaluated left to right and there is no provision for parenthesized expressions. Arithmetic is carried out in signed twos-complement integer precision (16 bits MS-DOS, 32 bits on Amiga and MAC).

The following are legal expressions.

```
$FF/ Max_Char      (=25 decimal if  Max_Char was 10 decimal)
10+$FF/Max_Char
   (=26 decimal because 10 is added to 255 and the result, 265, is
                                                        divided by 10)
$ FF&Max_Char      (=10 decimal)
$ FF^Max_Char      (=$F5)
```

The following is legal but perhaps not what you want.

```
BigBuf EQU 250
BufPtrInc EQU  BigBuf   /   Max_Char
```

You may have expected BufPtrInc = 25 but its value will be 250 because "/ Max_Char" is considered a comment. White space is not permitted in an expression.

5.0 PSEUDO OPCODES

Pseudo Opcodes must start in other than the first column, like other mnemonics.

```
<white space> <Pseudo-Op> [Operand(s)]
```

5.1 List of Pseudo Opcodes

The following Pseudo Opcodes are supported.

```
ORG, FCC, FDB, FCB, EQU,= , RMB, BSZ, ZMB, FILL
PAGE, INCLUDE, END, OPT, IFD, IFND, ELSE, ENDIF
CODE, DATA, BSS, RAM and AUTO.
```

Note: Fxx, ZMB, BSZ, FILL pseudo- ops advance the PC and add bytes to the S record. They are counted in the "total bytes" issued at the end of assembly. RMB advances the PC only and is not counted in the "total bytes." However, RMB does cause the current S record line to terminate and begin a new line.

5.2 Symbol Value Assignment, Equivalence

```
<label>[:] EQU <*| constant|symbol|expression>
```

EQU may be used to set a symbol to the value of the Operand which may be a constant, symbol or expression. EQU causes the value of the PC to be over printed by the value of the label (symbol) in the listing for convenience. It does not affect the PC. "=" is an alternative for "EQU."

Examples:

```
BigBuf EQU 250
BufPtrInc EQU  BigBuf/Max_Char  ;with no white space
StartPtr = $0A+BufPtrInc
```

However, EQU * sets the label to the value of the current PC as in:

```
Start EQU $E000
   ORG Start      ;set PC to $E000, the value of Start
   MADD EQU *     ;the value of MADD is $E000
   FADD EQU *
```

A redefinable symbol may appear on both sides of the EQU pseudo-op:

```
X@ EQU $FF        ;X@ = 255 decimal
    ldaa #10
    staa X@       ;stored 10 at address 255
X@ EQU X@-5       ;X@ = 250 decimal
    staa X@       ;stored 10 at address 250
```

A symbol is defined from the point in the source file where it is first used as a symbol with EQU (or alternately where the symbol is first used as a label). Setting a redefinable symbol to value zero does not " undefine" it. All symbols are logically "undefined" at the beginning of each pass but may well carry a value from one pass to the next.

Caution: Equates for other than redefinable symbols that have forward references cause Phasing Errors in Pass 2.

5.3 Memory Allocation & Definition

ORG, FCC, FDB, FCB, RMB, ZMB, FILL

ORG <constant | symbol | expression>

ORG (originate) sets the PC (program counter) to the value of the operand. There are no restrictions on how many times or where you may ORG. The assemblers will not stop you from ORGing into address space that doesn't exist on your processor or that doesn't make sense for the next operations.

Examples:

```
Y_ROMAN EQU $E000
   ORG Y_ROMAN    ;this is my personal favorite, sets PC=$E000
   ORG $E000      ;sets PC=$E000
ASM EQU $E000
   ORG ASM+10     ;this is a close second, sets PC=$E00A
```

Caution: ORGing without properly defining the segment (CODE, DATA etc. if you use these) may cause phasing errors on pass 2.

FCC '<ASCII characters>'

FCC (form constant characters) is used to sequentially place byte values of a string of one or more ASCII characters into memory. Value means the ASCII code itself. The value is that of the host system and so may vary if not the standard alpha- numerics. The delimiter used is your choice. A ' character is shown above but the assemblers simply take the first character in the string as the current delimiter which must appear again at the end to avoid an error. So the format is actually:

FCC < delim> <ASCII characters> < delim>

Examples:

```
FCC 'No funds available!'
FCC "Take all the money you want."  ;much better
FCC  dNOT AN  ERRORd
```

The first statement would produce the following listing if PC = $E000

```
E000 4E 6F 20 66 75 6E
64 73 20 61 76 61
69 6C 61 62 6C 75
21
```

A maximum of six hex bytes are printed per line in the listing, the rest appearing on lines below.

FDB <word>[,<word>] [,...etc.]
where <word> is < constant|symbol|expression>

FDB (form double bytes) is used to fill the memory with "long words" or 16 bit quantities. One could generate long word data tables or more likely, jump vector tables. A word of data is optionally followed by another using ',' as delimiter. Spaces are not permitted but will not cause an error during assembly, just termination of FDB.

Examples:

```
FDB 1             ;fills word with $0001
FDB $E000,25,@674 ;mix constants
FDB Start,Start+10,Start+20
```

The following does not cause and error but does not do what you want.

```
FDB $E000  ,$E000+10
;just fills memory with $E000 skipping next term
```

FCB <byte> [,<byte>] [,...etc.]
       where <byte> is < constant|symbol|expression>

FCB (form constant bytes) is used to fill the memory with "short words" or 8 bit (byte) quantities. It's useful in the construction of data tables. A byte is optionally followed by another using ',' as delimiter. Spaces are not permitted in the data but will not cause an error, just termination of FCB.

Examples:

```
       FCB 1             ;fills byte with $01
       FCB $FF,2,@17
       FCB 'A,'B,'C|$80,'D,'E|$80  ;parity table
```

The following does not cause an error but skips all but the first term.

```
       FCB $FF  ,2,@17
```

RMB <constant|symbol|expression>

RMB (reserve memory bytes) is useful to mark RAM locations as being the beginning of buffers, temporary tables, variables etc. It advances the PC but does not place any data in the S record. The Operand contains the number of bytes to advance the PC.

Examples:

```
SizeOfBuf    EQU 128            ;128 characters  max
Buf:         ORG $10            ;Input buffer
             RMB SizeOfBuf      ;make buffer (advance 128 characters)
             NOP                ;PC=$90 for this instruction
Buf2:        RMB SizeOfBuf*2    ;Buf2 is 256 l ong, PC=$91
             NOP                ;PC=$191
             RMB 1000           ;big RAM! Advances PC by 1000 decimal
Scratch:     RMB 1              ;one byte
```

ZMB <constant | symbol | expression>

BSZ <constant | symbol | expression>

ZMB (zero memory bytes) is similar to RMB except that in addition to advancing the PC, zero value bytes are written to the S record and would appear in a ROM if the S record is turned into firmware. Examples would be similar to RMB. BSZ (block store zeros) has the identical function. The user is left to think of a use for this pseudo-op.

```
FILL <value>,<byte_count>
       where <value> is <constant | symbol | expression>
                     < byte_count> is <* | constant | symbol | expression>
```

FILL can be used to fill memory with something other than zeros. Beginning at the current PC, memory is filled with < byte_count> number of bytes having <value>. Spaces are not permitted in the operand. * stands for the current value of the PC (be careful; could also be multiply depending upon context).

Examples:

```
       FILL $A5,$10      ;ROM check ($A5 is repeated 16 times)
TopMem EQU $FFE0         ;highest memory location used short of jump
                                                          vectors
       FILL  CheckSymbol,TopMem-*  ;fill out the rest of the ROM with a
                                                          given byte
```

In the example above, CheckSymbol would fill up to but not including location TopMem. The expression (TopMem-*) computes the number of bytes left before TopMem by subtracting the current value of the PC indicated by '*'.

5.4 Conditional Assembly and Include Files

```
INCLUDE <file>
```

Include files have the following format:

```
          include < path:filename>  ;comment
       or
          INCLUDE " path:file name" ;comment
               ...
               (include file lines)
               ...
               end
```

The pseudo op (key word) INCLUDE may be upper or lower case, beginning in any column except column one (reserved for labels). An included file name must be surrounded by either < > or " " (brackets or quotes). A full path name must be given; there is no option -<search path>. The path and file names may each contain spaces or any printing character but this may not be permitted except on the Amiga. AmigaDOS and most others permit search paths to be assigned in the calling script or batch file. Another method is to always use the fictitious path name " assem_lib:" and ASSIGN assem_lib: <path> in the calling script (AmigaDOS only). All DOS will search the current directory first (unless re-set by user). A comment may follow the include statement. Any delimiter or even none is acceptable but use ; or * for future compatibility with semi-colon preferred because of its use in other cross assemblers.

Each INCLUDE file must end with the pseudo op (key word) "end" (without quotes). Each of the files given as command line arguments may finish with "end" and it is best to do so as a matter of form. It has no effect, however, and assembly will continue after that point with the next command line file whereas assembly of an include file will terminate at "end."

The maximum number of nested include files open at one time is 30 including the top level. The number of include files may be as large as the system permits but they may not nest any further than 30 deep. Note: The assembler does not deallocate memory used to store include file names until the assembly process is finished. This could be a problem on MS-DOS systems especially if very many include files are used. Command line file names are not deallocated either but they may be handled differently. Motorola 68000 series base PCs do not have these problems.

IFD/IFND/ELSE/ENDIF

Conditional assembly is possible using IFD (if defined) and IFND (if not defined) pseudo operands and any symbol. The test determines only if the mentioned symbol has been defined in an active block of code and does not test or affect the value. The symbol value may be zero or any other value and the symbol may or may not be used as a replacement symbol but it will still test TRUE as IFD <symbol> if it has been defined at least once before the test. Using the symbol as an operand does not define it. It must appear beginning in column 1 of a line (label) for definition.

An active block of code is used by the assembler for code generation. An active block is one which appears outside of an IF statement or is between two IF/ELSE/ENDIF statements which test TRUE. The format is:

```
IF[ND] <symbol>

; code, more  IFs, comments and definitions

ELSE

; code, more  IFs, comments and definitions

ENDIF
```

The rules are:

1) each IF[ND] must be matched with a corresponding ENDIF.

2) ELSE causes a change of state from active to inactive or vice versa if part of an active IF[ND] block.

3) IF[ND] blocks contained within an inactive block are defined as inactive in their entirety. ELSE does not cause a change of state.

4) The matching ENDIF for an IF[ND] must occur in the same source file

Examples:

#1

```
    ifnd sysdat    ; ifnd may be upper or lower case
sysdat:            ;define so that other files with same
                   ;data will not call same include file
   include < sysdat.i>
                    ;system constant declarations
    endif
```

#2

```
    ifnd foo
fee: EQU 2         ;this block is active (no   foo before)
foo:               ;defining  foo
   LDAA #2         ;and is assembled to S record
   NOP
    ifd fee
                    ;fee was defined above
   STAA temp2      ;this block also active
   NOP
   else
   STAA temp1      ;this block not active & not assembled
    endif
   LDAB serial     ;this block active
    endif
   STAB buffer     ;this is always active & outside all if statements
```

5.5 Listing Control

Also see section 3.2 for command line options with the same effect.

PAGE

The pseudo-op PAGE causes a form feed and new page heading to be inserted into the listing output. No operand is needed or accepted. You may also use the "p50" option (command line or OPT) to automatically page the listing. PAGE is still useful for placing include files etc. at the top of a new page.

OPT <option>

The OPT pseudo-op allows the following operands:

nol	Turn off output listing
l	Turn on output listing (default)
noc	Disable cycle counts in listing (default)
c	Enable cycle counts in listing (clear total cycles)
contc	Re-enable cycle counts (don't clear total cycles)
cre	Enable printing of a cross reference table
s	generate a symbol table
crlf	enable <CR,LF> (non MS-DOS systems)
nnf	number include files continuous with command line files
p50	turn on page break every 50 lines approximately

Examples:

```
       OPT p50           ;turn on page breaks
       IFND Expand
       OPT  nol           ;disable include file listing
       ENDIF
       include < data.i>
       OPT l             ;turn listing back on
```

5.6 Program Counter Control (Segment)

CODE

DATA

BSS

RAM

AUTO

PC control pseudo operands save and restore the PC effectively assigning different code and data segments to different physical portions of the memory. CODE, DATA, BSS and AUTO switch the assembler between four different program counters which each may have their own ORG statements. RAM is an equivalent for BSS (block storage segment). Although equivalent in every way, the names are intended to convey the physical concepts of CODE (ROM), DATA (ROM), static variable RAM and dynamic / scratch or AUTOmatic ram. Typically, the user sets each PC to an initial value in a global data file. E.g.

```
CODE
ORG $E000         ;set start of program ROM
DATA
ORG $F800         ;set start of data section of ROM
BSS
ORG $0000         ;set start of RAM
```

Then, as files are added or deleted during development or for different configurations, the programmer marks each segment according to its purpose and the assembler fills that physical memory from the last entry. Normally, RMB statements only would occur in a BSS, RAM or AUTO segment for the purpose of alloting RAM. AUTO is suggestive of use as a scratch data PC and might be ORGed in every routine. However, the names are only suggestive and may be used as the programmer desires.

5.7 Ignored Pseudo- Opcodes

These Pseudo Opcodes are ignored (cause no action including error).

```
SPC Use blank lines instead
TTL use `pr' to get headings and page numbers
NAM[E] Did you ever use this one anyway?
```

6.0 TARGET MACHINE SPECIFICS

(as0) 6800:

Use for 6802 and 6808 too.

(as1) 6801:

You could use this one for the 6800 and avoid LSRD, ASLD, PULX, ABX, PSHX, MUL, SUBD, ADDD, LDD and STD.

(as4) 6804:

The symbols 'a', 'x' and 'y' are predefined as $FF, $80 and $81 respectively. Also defined as 'A', 'X' and 'Y'. Because of the 6804 architecture, this means that ' clr x' will work since the x register is just a memory location. To use short-direct addressing, the symbol involved must not be a forward reference (i.e. undefined) and must be in the range $80-$83. Remember that bytes assembled in the range $10-$7F will go into the data space; There is no program space ROM for these locations.

The syntax for Register indirect addressing is as follows:

mnemonic [<x>or<y>]

an example is:
lda [x]
the comma ',' is not allowed.

The MVI instruction (move immediate) has its own format :

mvi address,#data
where address is an 8-bit address in page zero, and data is the value to be written to
specified location.

(as5) 6805:

There is no 'opt cmos' pseudo, so be careful not to use STOP or WAIT in a program that is destined for an NMOS version of the 6805. The MUL instruction should also be avoided on all versions of the 6805 except the C4. Cycle times are for the NMOS versions.

(as9) 6809:

The SETDP pseudo-op is not implemented. Use the '>' and '<` operators to force the size of operands.

For compatibility, CPX is equal to CMPX.

(as11) 68HC11:

Bit manipulation operands are separated by blanks instead of commas since the 'HC11 has bit manipulation instructions that operate on indexed addresses.

7.0 LIMITATIONS AND ERRORS

7.1 File and Symbol Table Size

Source code and output data are stored in memory only line at a time only. Therefore, there is practically no limitation on the size of the source code except that imposed by the file storage media itself. Symbol definitions and values are stored in memory. MS-DOS has room for somewhat less than 2000 symbols of eight characters (the data requirements have been increased over previous versions). Amiga and MACs are almost always 1 Meg RAM or larger and so will have enough room for more than 5000 with room to spare for other data and the O/S. MAC II/X and Amiga 2500 are essentially unlimited.

The maximum number of nested include files is 30 (at one time).

7.2 Symbol Names

The maximum length of each symbol name is 15 characters.

7.3 Lost and Locked Files

Each source file is opened for reading and closed as required. Each file is read twice, once each pass. It's possible that the media might be removed or the file be otherwise made unavailable on the second pass. This would generate a file opening error. Restart the assembly process if this happens (find the lost file first!). With the Amiga and certain multi-tasking versions of MS-DOS (windows), the user can create problems by editing files during assembly. The editor may place a file lock on a source code file (MEMACS doesn't) causing a file open error. Really strange things can happen if the user changes a file between first and second assembly passes which might not cause an assembly error notice. On some systems, including the Amiga, you may be able to prevent this by protecting and un-protecting the file using a script which also calls the assembler (see "protect" and (D)OS manuals). File locks are not implemented in the assembler for cross system compatibility.

An assembler crash may also leave a file lock on one of the source files. See your (D)OS manual if this happens. Back up the source files!

7.4 Phasing Errors

A phasing error is reported if the value of a symbol is not the same on the second pass as it was on the first pass. A forward reference using EQU will cause a phasing error. However, no value checking is performed for redefinable symbols. Be careful using redefinable symbols.

Bad Example:

```
BufPtr EQU  NextVal       ;but  NextVal is not defined yet
...
NextVal EQU $A000
...
```

On the first pass, NextVal would be evaluated to zero and saved in BufPtr. On the second pass NextVal would be $A000 which would not match the value stored in BufPtr.

7.5 Error Listings

Error diagnostics are placed in the listing file just before the line containing the error. Format of the error line is:

<file name>,Line_number: Description of error

Errors in pass one cause cancellation of pass two. Error messages are meant to be self-explanatory. The file name precedes the error complete with path.

The following is a listing of errors with some sparse comments where required. You can add more notes as you make the mistakes.

```
Auto Inc/Dec by 1 or 2 only
Auto Inc/Dec Illegal on PC
Bad fill                  need second operand or missing comma
Bit Number must be 0-7    bit set or clear, only 8 bits on the port!
Branch out of Range       can happen if label not defined (conditional
assembly?)
Can't do both!  predec &  postinc not both possible
Can't Pull S from S
Can't Pull U from U
Can't Push S on S
Can't Push U on U
EQU requires label
Error: ELSE without IF                improper  nesting?
Error: ENDIF without IF               improper nesting?
Error: IFD/IFND nested too deep       maximum of 30 include files active
Extended Addressing not allowed
Illegal Register for Indexed
                         -this register doesn't have indexed addressing
Illegal Register Name
Illegal Symbol Name                   first character must be letter
```

```
Immediate Addressing Illegal
Immediate Operand Required
Indexed Addressing Required
Missing ']'
Missing ,
Missing Delimiter              mismatching  or no delimiter for FCC
No Auto  Inc/Dec by 1 for Indirect
Offset must be Zero
Operand must be $80 or $81
PCR illegal here
Phasing Error                  did EQU with forward reference?
Register List Required
Register Name Required
Register Size Mismatch
Symbol Redefined               only symbols ending in '@' may be redefined
Symbol table full  -out of memory - try killing RAM: files or stopping
                                                                  other tasks
Symbol undefined Pass 2        need at least one EQU or use as label
Syntax    6811 - got lost reading indexed addressing or    bset/bclr - see manual
Undefined Operand during Pass One
Unknown Addressing Mode
Unrecognized Mnemonic          spelling?
Unrecognized OPT               spelling?
```

7.6 Warning Listing

<file name>,Line_number: Warning --- Description of error

Warnings do not cause cancellation of pass two but should cause you to wonder where they came from.

```
Indexed Addressing Assumed
Missing ','
Missing ']'
No symbol for IFD        presumed undefined
No symbol for IFND       presumed undefined
Value truncated          FCB needs value <=255 or other operand too big
warning: can't open INCLUDE file <file>
                   May not be an error depending upon usage. See section 10.
```


_

7.7 Fatal Error Listing

Some errors are classed as fatal and cause an immediate termination of the assembly. These can be caused by loss of or mysterious change in a source code or forward reference file.

Pseudo error — real error - assembler is very lost in pseudo ops
Error in Mnemonic table — assembler very lost in Mnem onic table
bitop — addressing mode may not accept bit manipulation
Can't create temp file — can't make file Fwd_ref; protected? disk out? Forward ref file has gone
remove disk? lock file? Opened by another process or deleted?
Can't create object file — previous S19 file protected? disk out?
Error: file ended before IFD/IFND/ELSE/ENDIF -no matching ENDIF for all IF[ND]
Can't get here from there — assembler very lost in conditional assembly
Fatal Error: file ended before last ENDIF -can't carry conditional statements to next file

8.0 HOST SPECIFICS

8.1 MS-DOS PCs

The assemblers are complete executable machine code files ending in *.EXE. If using a hard disk system, it may be most convenient to place the assemblers in use in the DOS directory which is probably C:\DOS so that it's not necessary to change the search path each time you want to use them. Regardless, the path should be set to the assemblers or the assemblers should be in the working directory. PATH=C:\DOS is an example. You can show the search path by typing just PATH.

Command line operation is exactly as explained above. You must use file indirection to produce printed or saved listings.

```
AS11 PROG.ASM SUB1.ASM SUB2.ASM -L CRE S   >PRN:
```

Will assemble Prog, Sub1 and Sub2 to Prog.S19 with listing, cross reference and symbol table output to the printer. You may replace PRN: with the name of a file such as C:\WORK\DUMP.TXT and then COPY C:\WORK\DUMP.TXT PRN: at a latter time. A batch file will be much more convenient for the long assembly command line above. MS-DOS 3.x batch files will accept command line arguments making the batch files much more flexible. You can read the listing more conveniently at the CRT console if you use a "pipe" to "more" which should be in the search path (C:\DOS?). E.g. "as11 myprog.asm -l | more" where '|' is the pipe command. Hit <return> for the next screen. The assemblers are interruptable (^C or <control> <break>) if anything goes wrong rather than watch a long, useless listing.

If you do not have a hard disk system, make sure that there is enough room on the default floppy disk for FWD_REF and *.S19 files and that the write protect is off.

8.2 Apple MAC

Not available yet.

8.3 Commodore Amiga

The assemblers have no extension and are just as* such as as11. The Amiga is multi-tasking. Memory can vanish on a small system if you start many applications at once. The icon file supplied, as*.info is just for convenience in moving the files about using WorkBench. The assemblers will start by double clicking the icon but will shut down immediately because they will find no command line arguments. The assemblers are not written to accept messages from WorkBench. However, there are two other ways to run transportable code on the Amiga. What follows presumes use of AmigaOS 1.3 or higher.

Shell/CLI

The CLI (Command Line Interface) window is a character based console similar to MS-DOS. The command line arguments shown in this document will work as shown. Shell is an improved version of CLI virtually identical to UNIX "C Shell" or " Korn Shell." All CLI commands will work in Shell since as in UNIX they are just files in the command directory. It may be convenient if you copy the assemblers to directory c: (command) making them just another AmigaDOS command. Otherwise it may be best to "protect as11 wd-" for example to eliminate the possibility of over writing or deleting them. Alternately, make the assembler a resident DOS command using "Resident < path:assembler>" It will be added to AmigaDOS. Do not set the "pure" bit if you make the assembler resident. File and path names should use AmigaDOS/UNIX conventions. Long file and path names with embedded spaces are permitted. Use command line quotes around files having spaces: as11 "New Program.asm" -l.

Unfortunately, the Amiga assemblers are not interruptable from the dispatching Shell. It's preferred that you have two Shells open and that if an assembly is producing unbreakable garbage, you go to the second Shell and continue until the first finishes. Use " ChangeTaskPri 1" to get a higher priority than the Shell running the failed assembler. Use the RAM: disk to speed up floppy disk systems. If you use floppy disks, make sure that the default directory is on a disk which is not write protected and that *.S19 in use and Fwd_ref are not protected.

File indirection on the Amiga requires that indirection be given immediately after the command. E.g. "as11 > prt: MyProg.asm -l cre". You can substitute any device or file for prt:, even "speak:" if you want to have the listing read aloud whilst your working on the bench! It's not necessary to use "more" on the Amiga to stop and read a console listing. Just hit space to stop the listing and <back space> to start it again. However, you may pipe: to More if you want or redirect to ram:dump and read that with More. There can be more than one pipe:, Shell and assembly going at once.

Scripts and assembler source code editing can be done with MEMACS included on the Utilities disk. MEMACS is virtually identical to EMACS available on UNIX and described in most UNIX manuals. You can also assign directories new temporary names, especially useful with a hard disk system.

Workbench:

You can also get the assemblers and all your scripts to operate from Workbench by copying an icon to the script or writing them with NotePad (must be in some visible directory then like your work directory and not s:). Use WorkBench info to reset the default tool to Iconx (Icon Execute). Double clicking the script icon will then execute the script, possibly assembling your program with new version extension, protecting it and sending it down to the EVM or PROM burner in one swell fwoop. The window that opens up is a NEWCON: or Shell type window which accepts the <space bar> <back space> scroll commands. The process behind the window is IconX running execute. You can also get WorkBench to pass command line arguments to a script file by using Iconx. However, IconX can be a little buggy handing off more than one argument so it is much better to use Include files to pick up other source code. It should still be possible to assemble subroutines etc. using conditional assembly features to pick up data as needed and thus work strictly from WorkBench using the assemblers. An example script is:

```
.Key file ;accepts one file
echo "Assembling with listing"
as11 <file>  -l s   ;this is just command line as before
```

Type in the above using NotePad (supplied with Amiga) and save in your source code directory (perhaps as " DoIt"). Copy " DoIt.info" to source code if necessary to give them icons as in "copy DoIt.info to Prg1.asm.info." This will leave an Icon for the assembler script and source code as well. Select the " DoIt" script Icon and use " info" from Workbench. Then change the default tool to " c:IconX" and possibly delete Tool Types as well changing it to some thing like "WINDOW=NEWCON:10/10/620/150/Assembler". Select SAVE. You can pass a command line file to as11 by holding down the shift key (extended selection) and selecting DoIt first then the assembly (*.asm) file, double clicking the *. asm file. Consult the AmigaDOS and Enhancer 1.3 manuals. If you use RAM: IconX will pass the name RAM DISK which will get fragmented into two names; RAM (without colon) and DISK: (with colon). You may want to "assign DISK: RAM:" in the above script. The assembler will announce that it did not find file RAM but will go on to discover all the source code anyway. Avoid using directory names with spaces if working through IconX.

You can get editors that assign Icons automatically (MEMACS doesn't) and that will permit you to execute scripts from within a window making Motorola free ware cross assemblers a part of an "integrated" package.

9.0 TABLE FORMATS

9.1 Symbol Table

The format of the symbol table is:

```
     <label> <value>
```

where <label> is the symbol name in ascending, alphabetical order and <value> is the value of that symbol in four place hex notation. The value listed for redefinable symbols is the last value assigned.

9.2 Cross Reference Table

The format of the cross reference table is:

```
     <label> <value> *< definition_line> [ <line1>] [ <line2] [ ...etc.]
```

where <label> is the symbol name in ascending, alphabetical order and <value> is the value of that symbol in four place hex notation. The value listed for redefinable symbols is the last value assigned. The line number marked * is that line where the symbol was defined. Subsequent line numbers indicate where that symbol was used as an operand alone or in an expression. Subsequent definitions of redefinable symbols will not be listed.

Line numbers are those appearing in the listing. If include files are numbered separately (default), then the line numbers refer to that include file. Use command line option or OPT nnf to obtain sequential line numbers if you do not want to guess which file the symbol is in.

9.3 Listing

The listing format is:

```
Assembler Release <release> <version>   (once at beginning of assembly)
(c) Motorola (free ware)

                    <file name> <page number>     (each new page)

[error message]
<address> <code>    <source line no.> <source line>
...
...
Program +  Init Data = <byte size>
Error count = <no. of errors>
```

Error messages precede the line containing the error. Program + Init data is the number of bytes placed in the S record. If this value is zero but there are no errors, check conditional assembly statements to see if the entire program was conditionally not assembled.

10.0 SOME TECHNIQUES

Most programmers now use top down structured design. It may be data or procedurally oriented, or a combination of both; object oriented. Any of these techniques breaks software design and coding into smaller, more manageable modules. One would like to assemble and test the modules as they are created. Obviously a linking, macro-assembler would be best, but most of the features required are present in this assembler, especially for the first two techniques.

Procedures / functions / subroutines can be assembled and tested apart from any other if there is a mechanism to automatically incorporate global data and drivers when assembling subroutines only (bottom up coding). For top down coding, there should be a way to include the actual subroutines as they are developed and dummy routines if they are not. Static and dynamic (auto) data local to the subroutine should be allocated without conflict with other routines taking into account memory partitioning in a (possibly) firmware system.

Conditional assembly and include files may be used to control assembly as software is developed. One technique is to define and test a symbol having the same name as the file(s) to be included. E.g. if Global_Data is not defined, then include < Global_Data.i>. Define the symbol Global_Data within that file so that succeeding files which require that data will not pick it up and cause a redefine error. You may want to include a subroutine if it exists and a dummy subroutine if it does not. For this reason, a failure to find an include file is a warning and not an error (klutzy way of doing IF EXISTS). For example, you could include <Output.asm> and then test symbol OUTPUT which would be defined in Output.asm. IFND OUTPUT, then include < Dummy_Out.asm> (or something with a shorter name on MS-DOS).

Static data and code can be added in sequence to the memory type required (RAM or ROM) by using the pseudo- ops CODE, DATA and BSS (or RAM). These three (segment) memory types should be set once using followed by ORG <address>. After that, use just and the PC will be set to the next available location in the appropriate area. A naming convention can be used in place of local symbols. E.g. msg.main or msg_m for a symbol in routine Main.asm.

Auto or dynamic memory (sometimes just called scratch memory) is more difficult. Some dynamic storage is available using the stack but that is often not enough or convenient. You could use names such as Scratch1, Scratch2 etc. throughout, possibly setting local symbols equivalent as needed. However, the AUTO segment PC is intended to suggest its use as a local automatic or dynamic data PC. That is, use AUTO followed immediately by ORG <address> to the area you have established as reusable or scratch. Then immediately list the local symbols using RMB statements.

If you need more PCs than the four provided, you can manufacture more by saving the current value with e.g. TempPC@ = * which will assign the current PC value to the redefinable symbol (variable) TempPC@. You may then ORG <address> and continue. When you are through, save your PC in another variable and recover the old thus: ORG TempPC@.

Example

The following example is a demonstration of the assembler techniques discussed above using AS11 assembler (not of a working piece of code). It assembles properly; it has never been tested. The command line was "AS11 Main.asm -l cre". Files Global_Dat.i and Dum_Output.asm were picked up via include statements. Global_Dat.i is needed by both Main and Dum_Output assembly files. It's included conditionally so that if Dum_Output.asm alone is assembled, it receives the global data. Output.asm is the name of the final Output routine, not yet written.

```
Assembler release TER_2.0 version 2.09
(c) Motorola (free ware)
Main.asm, line no. 42:  warning:can't open INCLUDE file  Output.asm
0001                        ;****************************
0002                        ;       Main.asm
0003                        ;
0004                        ;     Main program
0005                        ;****************************
0006
0007                         IFND GLOB_DAT
0000                         include <  Global_Dat.i>   ;pick up  globals
0001                        ;***********************************
0002                        ;       Global_Dat.i
0003                        ;
0004                        ;     Global Variables & Data
0005                        ;***********************************
0006
0007 0001                   GLOBAL_DAT EQU 1     ;to prevent further
                            inclusion
0008
0009                         OPT p50   ;page breaks
0010
0011                        ;Memory Map
0012
0013                         CODE
0014 E000                    ORG $E000 ;set CODE
0015                         DATA
0016 F800                    ORG $F800 ;set data
0017                         BSS
0018 0040                    ORG 64         ;set static variable RAM
0019
0020
0021
```

```
0022                          ;Global Definitions/Equates
0023
0024 0001                     BYTE EQU 1
0025 0002                     WORD EQU 2
0026 0002                     APTR EQU 2
0027 0000                     EOS  EQU 0 ;end of string
0028
0029                          ;Global Variables
0030
0031 0040                     Acc1: RMB WORD
0032 0042                     Acc2: RMB WORD
0033
Global_Dat.i
                                    page   2
0035                          ;Global Structures
0036
0037                           DATA ;set PC to data area
0038
0039                          ;Device control register tables
0040                          ;  each entry is   struct  Hware where:
0041
0042                          ; struct  Hware definition
0043 0000                      U_Limit EQU 0             ;upper limit
0044 0001                      L_Limit EQU  U_Limit+BYTE  ;lower limit
0045 0002                      HErr_Msg EQU  L_Limit+BYTE ;*high error message
0046 0004                      LErr_Msg EQU  HErr_Msg+APTR     ;*low error
                                                          message
0047 0006                     Handler EQU   LErr_Msg+APTR ;*handler
0048 0008                      SizeOfHware EQU  Handler+APTR
0049
0050                           Reg_Table: ; struct  Hware(s) declaration
0051 F800 FF 0A                FCB $FF,10     ;  struct  Hware Reg1
0052 F802 F8 19 F8 26 E0 27    FDB Vy_Low_Msg,High_Msg,Handler1
0053
0054 F808 64 00                FCB 100,0 ;  struct  Hware Reg2
0055 F80A F8 10 F8 26 E0 28    FDB Low_Msg,High_Msg,Handler2
0056
0057                          ;     etc.
0058
0059                          ;Global Data
0060
0061 F810 54 6F  6F 20 6C 6F   Low_Msg: FCC 'Too low!'
     77 21
0062 F818 00                   FCB EOS
0063 F819 57 61 79 20 74 6F    Vy_Low_Msg: FCC 'Way too low!'
     6F 20 6C 6F 77 21
0064 F825 00                   FCB EOS
0065 F826 57 61 79 20 74 6F    High_Msg: FCC 'Way too high!'
     6F 20 68 69 67 68
     21
0066 F833 00                   FCB EOS
0067
0008                           end
0009                           ENDIF
0010
```

```
0011                            CODE ;set PC to next code area
0012
0013 E000 B6 F8 34             Start:       ldaa Reg_dat.m ;data for hardware
0014 E003 F6 F8 35               ldab First.m   ;register #
0015 E006 BD E0 1C               jsr Output
0016
0017 E009 F6 F8 36               ldab Second.m  ; reg #
0018 E00C BD E0 1C               jsr Output
0019
0020 E00F 86 FF                  ldaa #$FF ;nonsense busy loop
0021 E011 B7 00  00              staa Scratch1  ;to use RAM
0022 E014 7A 00  00             Loop@:      dec Scratch1
0023 E017 26 FB                  bne Loop@
0024
0025 E019 7E E0 00               jmp Start ;get the cables, George

Main.asm                                      page   3

0026
0027                           ;local static data
0028
0029                            DATA      ;set PC to next data area
0030 F834 C8                    Reg_dat.m: FCB #200  ;register data
0031 F835 00                    First.m:   FCB 0          ; reg nos.
0032 F836 01                    Second.m: FCB 1
0033
0034                           ;local dynamic data
0035
0036                            AUTO
0037 0000                       ORG 0     ;auto area
0038
0039 0000                      Scratch1: RMB BYTE
0040 0001                      Scratch2: RMB WORD
0041
Main.asm, line no. 42:  warning:can't open INCLUDE file   Output.asm
0042                            include <  Output.asm>
0043
0044                            IFND OUTPUT
0000                            include <  Dum_Output.asm>
0001                           ;*********************************
0002                           ;     Subroutine Dummy Output
0003                           ;
0004                           ;     Send data where it belongs
0005                           ;*********************************
0006
0007                            IFND GLOBAL_DAT
0008                            include <  Global_Dat.i>
0009                            ENDIF
0009                            ENDIF
0010
```

```
0011                            CODE      ;get code PC back
0012
0013 E01C FE F8 00             Output:      ldx Reg_Table  ;get pointer to
                                           table
0014 E01F 36                     psha      ;save data
0015 E020 86 08                  ldaa #SizeOfHware ;find offset
0016 E022 3D                     mul
0017 E023 3A                     abx       ;add offset to pointer
0018 E024 32                     pula      ;recover data
0019 E025 6E 06                  jmp Handler,x  ; goto Handler
0020
0021                           Handler1:
0022                            ;do handling
0023 E027 39                     rts
0024
0025                           Handler2:
0026                            ;do handling of second sort
0027 E028 39                     rts
0028
0029                            AUTO ;  goto auto area

Dum_Output.asm                                    page   4

0030 0000                       ORG 0     ;same ORG   cuz same area
0031
0032 0000                      Itch1:     RMB BYTE
0033 0001                      Itch2:     RMB BYTE
0034
0045                            end
0046                            ENDIF
0047
0048                            end
Program +  Init Data = 96 bytes
Error count = 0

APTR       0002 *0026 0046 0047 0048
Acc1       0040 *0031
Acc2       0042 *0032
BYTE       0001 *0024 0044 0045 0039 0032 0033
EOS        0000 *0027 0062 0064 0066
First.m    F835 *0031 0014
GLOBAL_DAT 0001 *0007
HErr_Msg   0002 *0045 0046
Handler    0006 *0047 0048 0019
Handler1   E027 *0021 0052
Handler2   E028 *0025 0055
High_Msg   F826 *0065 0052 0055
Itch1      0000 *0032
Itch2      0001 *0033
LErr_Msg   0004 *0046 0047
L_Limit    0001 *0044 0045
Loop@      E014 *0022 0023
Low_Msg    F810 *0061 0055
Output     E01C *0013 0015 0018
Reg_Table  F800 *0050 0013
Reg_dat.m  F834 *0030 0013
```


_

```
Scratch1   0000 *0039 0021 0022
Scratch2   0001 *0040
Second.m   F836 *0032 0017
SizeOfHware 0008 *0048 0015
Start      E000 *0013 0025
U_Limit    0000 *0043 0044
Vy_Low_Msg F819 *0063 0052
WORD       0002 *0025 0031 0032 0040
```

11.0 UPDATES & ERROR REPORTING

Known bugs

o Subsequent redefinitions of redefinable symbols are not mentioned in cross reference table. Suppose records are kept in Fwd_ref but didn't check yet. No functional description of modules.

o Include file listings with separate numbering show wrong line number at turn-over. Caused by fact that print_line() is last function before going on to next line and line number has already been changed by INCLUDE or END pseudo op. Line after is correct.

o Amiga Iconx calling .KEY file1 file2 (etc.) script with tests for <file(n)>=NULL causes very bizarre error where every line beginning with 'A' is misinterpretted (not and Opcode). Presume that Iconx or possibly AmigaDOS or Intuition has some how screwed up incoming file but how? Doesn't happen with single .KEY file argument and can use <include> to cover tracks. Notice that Iconx calls commands direct if no .KEY but uses Execute command if .KEY statement. Process may not be inheriting input stream correctly or something in path extracting \ tA as command. Path is ->input.device->Iconx-> Execute->Assembler with AmigaDOS and Intuition setting up and passing messages via Exec (I think).

o Outside loop ENDIF statement is in listing twice. Difficult to fix because inside loop need to be printed even if not assembled. However, end of loop exits to main assembler routines which print current line, again.