ASxxxx Assemblers

and

ASLINK Relocating Linker

Version 2.0

August 1998

Preface

The ASxxxx assemblers were written following the style of several cross assemblers found in the Digital Equipment Corporation Users Society (DECUS) distribution of the C programming language. The DECUS code was provided with no documentation as to the input syntax or the output format. Study of the code revealed that the unknown author of the code had attempted to formulate an assembler with attributes similiar to those of the PDP-11 MACRO assembler (without macro's). The incomplete code from the DECUS C distribution has been largely rewritten, only the program structure, and C source file organization remains relatively unchanged. However, I wish to thank the author for his contribution to this set of assemblers.

The ASLINK program was written as a companion to the ASxxxx assemblers, its design and implementation was not derived from any other work.

I would greatly appreciate receiving the details of any changes, additions, or errors pertaining to these programs and will attempt to incorporate any fixes or generally useful changes in a future update to these programs.

Alan R. Baldwin
Kent State University
Physics Department

Kent, Ohio 44242
U.S.A.

http://shop-pdp.kent.edu/ashtml/asxxxx.htm
baldwin@shop-pdp.kent.edu
tel: (330) 672 2531
fax: (330) 672 2959

Abstract

The ASxxxx assemblers are a series of microprocessor assemblers written in the C programming language. This collection contains cross assemblers for the 6800(6802/6808), 6801(hd6303), 6804, 6805, 68HC08, 6809, 68HC11, 68HC12, 68HC16, 8051, 8085(8080), z80(hd64180), H8/3xx, and 6500 series microprocessors. Each assembler has a device specific section which includes: (1) device description, byte order, and file extension information, (2) a table of assembler general directives, special directives, assembler mnemonics and associated operation codes, (3) machine specific code for processing the device mnemonics, addressing modes, and special directives.

The assemblers have a common device independent section which handles the details of file input/output, symbol table generation, program/data areas, expression analysis, and assembler directive processing.

The assemblers provide the following features: (1) alphabetized, formatted symbol table listings, (2) relocatable object modules, (3) global symbols for linking object modules, (4) conditional assembly directives, (5) reusable local symbols, and (6) include-file processing.

The companion program ASLINK is a relocating linker performing the following functions: (1) bind multiple object modules into a single memory image, (2) resolve inter-module symbol references, (3) resolve undefined symbols from specified librarys of object modules, (4) process absolute, relative, concatenated, and overlay attributes in data and program sections, (5) perform byte and word program-counter relative (`pc` or `pcr`) addressing calculations, (6) define absolute symbol values at link time, (7) define absolute area base address values at link time, (8) produce Intel Hex or Motorola `S19` output file, (9) produce a map of the linked memory image, and (10) update the ASxxxx assembler listing files with the absolute linked addresses and data.

The assemblers and linker have been tested using DECUS C under TSX+ and RT-11, PDOS C V5.4b, and Symantec C/C++ V6.1/V7.2 under DOS/Windows 3.x/95. Complete source code and documentation for the assemblers and linker is included with the distribution. Additionally, test code for each assembler and several microprocessor monitors (ASSIST05 for the 6805, MONDEB and ASSIST09 for the 6809, and BUFFALO 2.5 for the 6811) are included as working examples of use of these assemblers.

1 The Assembler

1.1 The ASXXXX Assemblers

The ASxxxx assemblers are a series of microprocessor assemblers written in the C programming language. Each assembler has a device specific section which includes:

1. device description, byte order, and file extension information

2. a table of the assembler general directives, special device directives, assembler mnemonics and associated operation codes
3. machine specific code for processing the device mnemonics, addressing modes, and special directives

The device specific information is detailed in the appendices.

The assemblers have a common device independent section which handles the details of file input/output, symbol table generation, program/data areas, expression analysis, and assembler directive processing.

The assemblers provide the following features:

1. Command string control of assembly functions
2. Alphabetized, formatted symbol table listing
3. Relocatable object modules
4. Global symbols for linking object modules
5. Conditional assembly directives
6. Program sectioning directives

ASxxxx assembles one or more source files into a single relocatable ascii object file. The output of the ASxxxx assemblers consists of an ascii relocatable object file (`*.rel`), an assembly listing file (`*.lst`), and a symbol file (`*.sym`).

1.1.1 Assembly Pass 1

During pass 1, ASxxxx opens all source files and performs a rudimenatry assembly of each source statement. During this process all symbol tables are built, program sections defined, and number of bytes for each assembled source line is estimated.

At the end of pass 1 all undefined symbols may be made global (external) using the ASxxxx switch -g, otherwise undefined symbols will be flagged as errors during succeeding passes.

1.1.2 Assembly Pass 2

During pass 2 the ASxxxx assembler resolves forward references and determines the number of bytes for each assembled line. The number of bytes used by a particular assembler instruction may depend upon the addressing mode, whether the instruction allows multiple forms based upon the relative distance to the addressed location, or other factors. Pass 2 resolves these cases and determines the address of all symbols.

1.1.3 Assembly Pass 3

Pass 3 by the assembler generates the listing file, the relocatable output file, and the symbol tables. Also during pass 3 the errors will be reported.

The relocatable object file is an ascii file containing symbol references and definitions, program area definitions, and the relocatable assembled code, the linker ASLINK will use this information to generate an absolute load file (Motorola or Intel formats).

1.2 Source Program Format

1.2.1 Statement Format

A source program is composed of assembly-language statements. Each statement must be completed on one line. A line may contain a maximum of 128 characters, longer lines are truncated and lost.

An ASxxxx assembler statement may have as many as four fields. These fields are identified by their order within the statement and/or by separating characters between fields. The general format of the ASxxxx statement is:

```
[label:]  Operator        Operand         [;Comment(s)]
```

The label and comment fields are optional. The operator and operand fields are interdependent. The operator field may be an assembler directive or an assembly mnemonic. The operand field may be optional or required as defined in the context of the operator.

ASxxxx interprets and processes source statements one at a time. Each statement causes a particular operation to be performed.

Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user defined symbol table. The current location counter is used by ASxxxx to assign memory

addresses to the source program statements as they are encountered during the assembly process. Thus a label is a means of symbolically referring to a specific statement.

When a program section is absolute, the value of the current location counter is absolute; its value references an absolute memory address. Similarly, when a program section is relocatable, the value of the current location counter is relocatable. A relocation bias calculated at link time is added to the apparent value of the current location counter to establish its effective absolute address at execution time. (The user can also force the linker to relocate sections defined as absolute. This may be required under special circumstances.)

If present, a label must be the first field in a source statement and must be terminated by a colon (`:`). For example, if the value of the current location counter is absolute `01F0(H)`, the statement:

```
abcd:     nop
```

assigns the value `01F0(H)` to the label `abcd`. If the location counter value were relocatable, the final value of `abcd` would be `01F0(H)+K`, where `K` represents the relocation bias of the program section, as calculated by the linker at link time.

More than one label may appear within a single label field. Each label so specified is assigned the same address value. For example, if the value of the current location counter is `1FF0(H)`, the multiple labels in the following statement are each assigned the value `1FF0(H)`:

```
abcd:     aq:     $abc:   nop
```

Multiple labels may also appear on successive lines. For example, the statements

```
abcd:
aq:
$abc:     nop
```

likewise cause the same value to be assigned to all three labels.

A double colon (::) defines the label as a global symbol. For example, the statement

```
abcd::    nop
```

establishes the label abcd as a global symbol. The distinguishing attribute of a global symbol is that it can be referenced from within an object module other than the module in which the symbol is defined. References to this label in other modules are resolved when the modules are linked as a composite executable image.

The legal characters for defining labels are:

```
A through Z
a through z
0 through 9
. (Period)
$ (Dollar sign)
_ (underscore)
```

A label may be any length; however, only the first 79 characters are significant and, therefore must be unique among all labels in the source program (not necessarily among separately compiled modules). An

error code(s) (m or p) will be generated in the assembly listing if the first 79 characters in two or more labels are the same. The m code is caused by the redeclaration of the symbol or its reference by another statement. The p code is generated because the symbols location is changing on each pass through the source file.

The label must not start with the characters 0-9, as this designates a local symbol with special attributes described in a later section.

The label must not start with the sequence $$, as this represents the temporary radix 16 for constants.

Operator Field

The operator field specifies the action to be performed. It may consist of an instruction mnemonic (op code) or an assembler directive.

When the operator is an instruction mnemonic, a machine instruction is generated and the assembler evaluates the addresses of the operands which follow. When the operator is a directive ASxxxx performs certain control actions or processing operations during assembly of the source program.

Leading and trailing spaces or tabs in the operator field have no significance; such characters serve only to separate the operator field from the preceeding and following fields.

An operator is terminated by a space, tab or end of line.

Operand Field

When the operator is an instruction mnemonic (op code), the operand field contains program variables that are to be evaluated/manipulated by the operator.

Operands may be expressions or symbols, depending on the operator. Multiple expressions used in the operand fields may be separated by a comma. An operand should be preceeded by an operator field; if it is not, the statement will give an error (`q` or `o`). All operands following instruction mnemonics are treated as expressions.

The operand field is terminated by a semicolon when the field is followed by a comment. For example, in the following statement:

```
label:    lda     abcd,x          ;Comment field
```

the tab between `lda` and `abcd` terminates the operator field and defines the beginning of the operand field; a comma separates the operands `abcd` and `x`; and a semicolon terminates the operand field and defines the beginning of the comment field. When no comment field follows, the operand field is terminated by the end of the source line.

Comment Field

The comment field begins with a semicolon and extends through the end of the line. This field is optional and may contain any 7-bit ascii character except null.

Comments do not affect assembly processing or program execution.

1.3 Symbols and Expressions

This section describes the generic components of the ASxxxx assemblers: the character set, the conventions observed in constructing symbols, and the use of numbers, operators, and expressions.

1.3.1 Character Set

The following characters are legal in ASxxxx source programs:

1. The letters `A` through `Z`. Both upper- and lower-case letters are acceptable. The assemblers, by default, are not case sensitive, i.e., `ABCD` and `abcd` are the same symbols. (The assemblers can be made case sensitive by using the `-z` command line option.)
2. The digits 0 through 9
3. The characters `.` (period), `$` (dollar sign), and `_` (underscore).
4. The special characters listed in Tables 1 through 6.

Tables 1 through 6 describe the various ASxxxx label and field terminators, assignment operators, operand separators, assembly, unary, binary, and radix operators.

Table 1: Label Terminators and Assignment Operators

:	Colon	Label terminator.
::	Double colon	Label Terminator; defines the label as a global label.
=	Equal sign	Direct assignment operator.
==	Double equal sign	Direct assignment operator; defines the symbol as a global symbol.

Table 2: Field Terminators and Operand Separators

	Tab	Item or field terminator.
	Space	Item or field terminator.
,	Comma	Operand field separator.
;	Semicolon	Comment field indicator.

Table 3: Assembler Operators

#	Number sign	Immediate expression indicator.
.	Period	Current location counter.
(	Left parenthesis	Expression delimiter.
)	Right parenthesis	Expression delimeter.

Table 4: Unary Operators

<	Left bracket	<FEDC	Produces the lower byte value of the expression. (DC)
>	Right bracket	>FEDC	Produces the upper byte value of the expression. (FE)
+	Plus sign	+A	Positive value of A
−	Minus sign	−A	Produces the negative (2's complement) of A
~	Tilde	~A	Produces the 1's complement of A
'	Single quote	'D	Produces the value of the character D
"	Double quote	"AB	Produces the double byte value for AB
\	Backslash	'\n	Unix style characters \b, \f, \n, \r, \t
	or	'\001	or octal byte values.

Table 5: Binary Operators

<<	Double Left bracket	0800 << 4	Produces the 4 bit left-shifted value of 0800. (8000)
>>	Double Right bracket	0800 >> 4	Produces the 4 bit right-shifted value of 0800. (0080)
+	Plus sign	A + B	Arithmetic Addition operator.
−	Minus sign	A - B	Arithmetic Subtraction operator.
*	Asterisk	A * B	Arithmetic Multiplication operator. (signed 16-bit)
/	Slash	A / B	Arithmetic Division operator. (signed 16-bit quotient)
&	Ampersand	A & B	Logical AND operator.
|	Bar	A | B	Logical OR operator.
%	Percent sign	A % B	Modulus operator. (16-bit value)
^	Up arrow or circumflex	A ^ B	EXCLUSIVE OR operator.

Table 6: Temporary Radix Operators

`$%`, `0b`, `0B`	Binary radix operator.
`$&`, `0o`, `0O`, `0q`, `0Q`	Octal radix operator.
`$#`, `0d`, `0D`	Decimal radix operator.
`$$`, `0h`, `0H`, `0x`, `0X`	Hexidecimal radix operator.

Potential ambiguities arising from the use of `0b` and `0d` as temporary radix operators may be circumvented by preceding all non-prefixed hexadecimal numbers with `00`. Leading `0`'s are required in any case where the first hexadecimal digit is `abcdef` as the assembler will treat the letter sequence as a label.

1.3.2 User-Defined Symbols

User-defined symbols are those symbols that are equated to a specific value through a direct assignment statement or appear as labels. These symbols are added to the User Symbol Table as they are encountered during assembly.

The following rules govern the creation of user-defined symbols:

1. Symbols can be composed of alphanumeric characters, dollar signs (`$`), periods (`.`), and underscores (`_`) only.
2. The first character of a symbol must not be a number (except in the case of local symbols).
3. The first 79 characters of a symbol must be unique. A symbol can be written with more than 79 legal characters, but the 80th and subsequent characters are ignored.
4. Spaces and Tabs must not be embedded within a symbol.

1.3.3 Local Symbols

Local symbols are specially formatted symbols used as labels within a block of coding that has been delimited as a local symbol block. Local symbols are of the form `n$`, where `n` is a decimal integer from 0 to 255, inclusive. Examples of local symbols are:

```
1$
27$
138$
244$
```

The range of a local symbol block consists of those statements between two normally constructed symbolic labels. Note that a statement of the form:

```
ALPHA = EXPRESSION
```

is a direct assignment statement but does not create a label and thus does not delimit the range of a local symbol block.

Note that the range of a local symbol block may extend across program areas.

Local symbols provide a convenient means of generating labels for branch instructions and other such references within local symbol blocks. Using local symbols reduces the possibility of symbols with multiple definitions appearing within a user program. In addition, the use of local symbols differentiates entry-point labels from local labels, since local labels cannot be referenced from outside their respective local symbol

blocks. Thus, local symbols of the same name can appear in other local symbol blocks without conflict. Local symbols require less symbol table space than normal symbols. Their use is recommended.

The use of the same local symbol within a local symbol block will generate one or both of the `m` or `p` errors.

Example of local symbols:

```
a:      ldx     #atable ;get table address
        lda     #0d48   ;table length
1$:     clr     ,x+     ;clear
        deca
        bne     1$

b:      ldx     #btable ;get table address
        lda     #0d48   ;table length
1$:     clr     ,x+     ;clear
        deca
        bne     1$
```

1.3.4 Current Location Counter

The period (`.`) is the symbol for the current location counter. When used in the operand field of an instruction, the period represents the address of the first byte of the instruction:

```
AS:     ldx     #.      ;The period (.) refers to
                        ;the address of the ldx
                        ;instruction.
```

When used in the operand field of an ASxxxx directive, it represents the address of the current byte or word:

```
QK = 0

.word   0xFFFE,.+4,QK   ;The operand .+4 in the .word
                        ;directive represents a value
                        ;stored in the second of the
                        ;three words during assembly.
```

If we assume the current value of the program counter is 0H0200, then during assembly, ASxxxx reserves three words of storage starting at location 0H0200. The first value, a hexadecimal constant FFFE, will be stored at location 0H0200. The second value represented by .+4 will be stored at location 0H0202, its value will be 0H0206 (= 0H0202 + 4). The third value defined by the symbol QK will be placed at location 0H0204.

At the beginning of each assembly pass, ASxxxx resets the location counter. Normally, consecutive memory locations are assigned to each byte of object code generated. However, the value of the location counter can be changed through a direct assignment statement of the following form:

```
. = . + expression
```

The new location counter can only be specified relative to the current location counter. Neglecting to specify the current program counter along with the expression on the right side of the assignment operator will generate the (.) error. (Absolute program areas may use the .org directive to specify the absolute location of the current program counter.)

The following coding illustrates the use of the current location counter:

```
            .area   CODE1   (ABS)   ;program area CODE1
                                    ;is ABSOLUTE

            .org    0H100           ;set location to
                                    ;0H100 absolute

num1:       ldx     #.+0H10         ;The label num1 has
                                    ;the value 0H100.
                                    ;X is loaded with
                                    ;0H100 + 0H10

            .org    0H130           ;location counter
                                    ;set to 0H130

num2:       ldy     #.              ;The label num2 has
                                    ;the value 0H130.
                                    ;Y is loaded with
                                    ;value 0H130.
```

```
            .area   CODE2   (REL)   ;program area CODE2
                                    ;is RELOCATABLE

            . = . + 0H20            ;Set location counter
                                    ;to relocatable 0H20 of
                                    ;the program section.

num3:       .word   0               ;The label num3 has
                                    ;the value
                                    ;of relocatable 0H20.

            . = . + 0H40            ;will reserve 0H40
                                    ;bytes of storage as will
            .blkb   0H40            ;or
            .blkw   0H20
```

The `.blkb` and `.blkw` directives are the preferred methods of allocating space.

1.3.5 Numbers

ASxxxx assumes that all numbers in the source program are to be interpreted in decimal radix unless otherwise specified. The `.radix` directive may be used to specify the default as octal, decimal, or hexadecimal. Individual numbers can be designated as binary, octal, decimal, or hexadecimal through the temporary radix prefixes shown in table 6.

Negative numbers must be preceeded by a minus sign; ASxxxx translates such numbers into two's complement form. Positive numbers may (but need not) be preceeded by a plus sign.

Numbers are always considered to be absolute values, therefore they are never relocatable.

1.3.6 Terms

A term is a component of an expression and may be one of the following:

1. A number.
2. A symbol:
   (a) A period (`.`) specified in an expression causes the current location counter to be used.
   (b) A User-defined symbol.
   (c) An undefined symbol is assigned a value of zero and inserted in the User-Defined symbol table as an undefined symbol.
3. A single quote followed by a single ascii character, or a double quote followed by two ascii characters.
4. An expression enclosed in parentheses. Any expression so enclosed is evaluated and reduced to a single term before the remainder of the expression in which it appears is evaluated. Parentheses, for example, may be used to alter the left-to-right evaluation of expressions, (as in `A*B+C` versus `A*(B+C)`), or to apply a unary operator to an entire expression (as in `-(A+B)`).
5. A unary operator followed by a symbol or number.

1.3.7 Expressions

Expressions are combinations of terms joined together by binary operators. Expressions reduce to a 16-bit value. The evaluation of an expression includes the determination of its attributes. A resultant expression value may be one of three types (as described later in this section): relocatable, absolute, and external.

Expressions are evaluate with an operand hierarchy as follows:

```
*       /       %       multiplication,
                        division, and
                        modulus first.

+       -               addition and
                        subtraction second.

<<      >>              left shift and
                        right shift third.

^                       exclusive or fourth.

&                       logical and fifth.

|                       logical or last
```

except that unary operators take precedence over binary operators.

A missing or illegal operator terminates the expression analysis, causing error codes (o) and/or (q) to be generated depending upon the context of the expression itself.

At assembly time the value of an external (global) expression is equal to the value of the absolute part of that expression. For example, the expression external+4, where 'external' is an external symbol, has the value of 4. This expression, however, when evaluated at link time takes on the resolved value of the symbol 'external', plus 4.

Expressions, when evaluated by ASxxxx, are one of three types: relocatable, absolute, or external. The following distinctions are important:

1. An expression is relocatable if its value is fixed relative to the base address of the program area in which it appears; it will have an offset value added at link time. Terms that contain labels defined in relocatable program areas will have a relocatable value; similarly, a period (.) in a relocatable program area, representing the value of the current program location counter, will also have a relocatable value.
2. An expression is absolute if its value is fixed. An expression whose terms are numbers and ascii characters will reduce to an absolute value. A relocatable expression or term minus a relocatable term, where both elements being evaluated belong to the same program area, is an absolute expression. This is because every term in a program area has the same relocation bias. When one term is subtracted from the other the relocation bias is zero.
3. An expression is external (or global) if it contains a single global reference (plus or minus an absolute expression value) that is not defined within the current program. Thus, an external expression is only partially defined following assembly and must be resolved at link time.

1.4 General Assembler Directives

An ASxxxx directive is placed in the operator field of the source line. Only one directive is allowed per source line. Each directive may have a blank operand field or one or more operands. Legal operands differ with each directive.

1.4.1 .module Directive

Format:

```
.module string
```

The `.module` directive causes the string to be included in the assemblers output file as an identifier for this particular object module. The string may be from 1 to 79 characters in length. Only one identifier is allowed per assembled module. The main use of this directive is to allow the linker to report a modules' use of undefined symbols. At link time all undefined symbols are reported and the modules referencing them are listed.

1.4.2 .title Directive

Format:

```
.title  string
```

The `.title` directive provides a character string to be placed on the second line of each page during listing.

1.4.3 .sbttl Directive

Format:

```
.sbttl  string
```

The `.sbttl` directive provides a character string to be placed on the third line of each page during listing.

1.4.4 .page Directive

Format:

```
.page
```

The `.page` directive causes a page ejection with a new heading to be printed. The new page occurs after the next line of the source program is processed, this allows an immediately following `.sbttl` directive to appear on the new page. The `.page` source line will not appear in the file listing. Paging may be disabled by invoking the `-p` directive.

1.4.5 .byte and .db Directives

Format:

```
.byte   exp             ;Stores the binary value
.db     exp             ;of the expression in the
                        ;next byte.

.byte   exp1,exp2,expn  ;Stores the binary values
.db     exp1,exp2,expn  ;of the list of expressions
                        ;in successive bytes.
```

where:

`exp`, `exp1`, ⋮ `expn`	represent expressions that will be truncated to 8-bits of data. Each expression will be calculated as a 16-bit word expression, the high-order byte will be truncated. Multiple expressions must be separated by commas.

The `.byte` or `.db` directives are used to generate successive bytes of binary data in the object module.

1.4.6 .word and .dw Directives

Format:

```
.word   exp             ;Stores the binary value
.dw     exp             ;of the expression in
                        ;the next word.

.word   exp1,exp2,expn  ;Stores the binary values
.dw     exp1,exp2,expn  ;of the list of expressions
                        ;in successive words.
```

where:

`exp`, `exp1`, ⋮ `expn` — represent expressions that will occupy two bytes of data. Each expression will be calculated as a 16-bit word expression. Multiple expressions must be separated by commas.

The `.word` or `.dw` directives are used to generate successive words of binary data in the object module.

1.4.7 .blkb, .blkw, and .ds Directives

Format:

```
.blkb   N       ;reserve N bytes of space
.blkw   N       ;reserve N words of space
.ds     N       ;reserve N bytes of space
```

The .blkb and .ds directives reserve byte blocks in the object module; the .blkw directive reserves word blocks.

1.4.8 .ascii Directive

Format:

```
.ascii  /string/
```

where:

string	is a string of printable ascii characters.
/ /	represent the delimiting characters. These delimiters may be any paired printing characters, as long as the characters are not contained within the string itself. If the delimiting characters do not match, the .ascii directive will give the (q) error.

The .ascii directive places one binary byte of data for each character in the string into the object module.

1.4.9 .ascis Directive

Format:

```
.ascis  /string/
```

where:

`string`	is a string of printable ascii characters.
`/ /`	represent the delimiting characters. These delimiters may be any paired printing characters, as long as the characters are not contained within the string itself. If the delimiting characters do not match, the `.ascis` directive will give the (`q`) error.

The `.ascis` directive places one binary byte of data for each character in the string into the object module. The last character in the string will have the high order bit set.

1.4.10 .asciz Directive

Format:

```
.asciz  /string/
```

where:

`string`	is a string of printable ascii characters.
`/ /`	represent the delimiting characters. These delimiters may be any paired printing characters, as long as the characters are not contained within the string itself. If the delimiting characters do not match, the `.asciz` directive will give the (`q`) error.

The `.asciz` directive places one binary byte of data for each character in the string into the object module. Following all the character data a zero byte is inserted to terminate the character string.

1.4.11 .radix Directive

Format:

```
.radix  character
```

where: `character` represents a single character specifying the default radix to be used for succeeding numbers. The character may be any one of the following:

`B`,`b`	Binary
`O`,`o`,`Q`,`q`	Octal
`D`,`d`	Decimal (default)
`H`,`h`,`X`,`x`	Hexidecimal

1.4.12 .even Directive

Format:

```
.even
```

The .even directive ensures that the current location counter contains an even boundary value by adding one if the current location is odd.

1.4.13 .odd Directive

Format:

```
.odd
```

The .odd directive ensures that the current location counter contains an odd boundary value by adding one if the current location is even.

1.4.14 .area Directive

Format:

```
.area   name    [(options)]
```

where:

name	represents the symbolic name of the program section. This name may be the same as any user-defined symbol as the area names are independent of all symbols and labels.

`options` specify the type of program or data area:

`ABS`	absolute (automatically invokes `OVR`)
`REL`	relocatable
`OVR`	overlay
`PAG`	concatenate
`CON`	paged area

The `.area` directive provides a means of defining and separating multiple programming and data sections. The name is the area label used by the assembler and the linker to collect code from various separately assembled modules into one section. The name may be from 1 to 79 characters in length.

The options are specified within parenthesis and separated by commas as shown in the following example:

```
.area  TEST  (REL,CON)  ;This section is relocatable
                        ;and concatenated with other
                        ;sections of this program area.

.area  DATA  (REL,OVR)  ;This section is relocatable
                        ;and overlays other sections
                        ;of this program area.

.area  SYS   (ABS,OVR)  ;(CON not allowed with ABS)
                        ;This section is defined as
                        ;absolute. Absolute sections
                        ;are always overlayed with other
                        ;sections of this program area.
```

```
        .area  PAGE  (PAG)      ;This is a paged section. The
                                ;section must be on a 256 byte
                                ;boundary and its length is
                                ;checked by the linker to be
                                ;no larger than 256 bytes.
                                ;This is useful for direct page
                                ;areas.
```

The default area type is REL|CON; i.e., a relocatable section which is concatenated with other sections of code with the same area name. The ABS option indicates an absolute area. The OVR and CON options indicate if program sections of the same name will overlay each other (start at the same location) or be concatenated with each other (appended to each other).

Multiple invocations of the .area directive with the same name must specify the same options or leave the options field blank, this defaults to the previously specified options for this program area.

The ASxxxx assemblers automatically provide two program sections:

'. .ABS.' This dummy section contains all absolute symbols and their values.

'_CODE' This is the default program/data area. This program area is of type (REL,CON).

The ASxxxx assemblers also automatically generate two symbols for each program area:

's_<area>' This is the starting address of the program area.

'l_<area>' This is the length of the program area.

The .area names and options are never case sensitive.

1.4.15 .org Directive

Format:

```
.org    exp
```

where exp is an absolute expression that becomes the current location counter.

The .org directive is valid only in an absolute program section and will give a (q) error if used in a relocatable program area. The .org directive specifies that the current location counter is to become the specified absolute value.

1.4.16 .globl Directive

Format:

```
.globl  sym1,sym2,...,symn
```

where:

sym1, sym2, ⋮ symn	represent legal symbolic names. When multiple symbols are specified, they are separated by commas.

A .globl directive may also have a label field and/or a comment field.

The .globl directive is provided to define (and thus provide linkage to) symbols not otherwise defined as global symbols within a module. In defining global symbols the directive .globl J is similar to:

```
J == expression or J::
```

Because object modules are linked by global symbols, these symbols are vital to a program. All internal symbols appearing within a given program must be defined at the end of pass 1 or they will be considered undefined. The assembly directive (-g) can be be invoked to make all undefined symbols global at the end of pass 1.

1.4.17 .if, .else, and .endif Directives

Format:

```
.if     expr
.                       ;}
.                       ;} range of true condition
.                       ;}
.else
.                       ;}
.                       ;} range of false condition
.                       ;}
.endif
```

The conditional assembly directives allow you to include or exclude blocks of source code during the assembly process, based on the evaluation of the condition test.

The range of true condition will be processed if the expression '`expr`' is not zero (i.e., true) and the range of false condition will be processed if the expression '`expr`' is zero (i.e false). The range of true condition is optional as is the `.else` directive and the range of false condition. The following are all valid `.if`/`.else`/`.endif` constructions:

```
.if     A-4             ;evaluate A-4
.byte   1,2             ;insert bytes if A-4 is
.endif                  ;not zero
```

```
.if     K+3             ;evaluate K+3
.else
.byte   3,4             ;insert bytes if K+3
.endif                  ;is zero

.if     J&3             ;evaluate J masked by 3
.byte   12              ;insert this byte if J&3
.else                   ;is not zero
.byte   13              ;insert this byte if J&3
.endif                  ;is zero
```

The .if/.else/.endif directives may be nested up to 10 levels.

The .page directive is processed within a false condition range to allow extended textual information to be incorporated in the source program without the need to use the comment delimiter (;):

```
.if     0

.page
This text will be bypassed during assembly
but appear in the listing file.
...

.endif
```

1.4.18 .include Directive

Format:

```
.include        string
```

where: `string` represents a delimited string that is the file specification of an ASxxxx source file.

The `.include` directive is used to insert a source file within the source file currently being assembled. When this directive is encountered, an implicit `.page` directive is issued. When the end of the specified source file is reached, an implicit `.page` directive is issued and input continues from the previous source file. The maximum nesting level of source files specified by a `.include` directive is five.

The total number of separately specified `.include` files is unlimited as each `.include` file is opened and then closed during each pass made by the assembler.

1.4.19 .setdp Directive

Format:

```
.setdp [base [,area]]
```

The set direct page directive has a common format in all the AS68xx assemblers. The `.setdp` directive is used to inform the assembler of the current direct page region and the offset address within the selected

area. The normal invocation methods are:

```
.area   DIRECT  (PAG)
.setdp
```

or

```
.setdp  0,DIRECT
```

for all the 68xx microprocessors (the 6804 has only the paged ram area). The commands specify that the direct page is in area DIRECT and its offset address is 0 (the only valid value for all but the 6809 microprocessor). Be sure to place the DIRECT area at address 0 during linking. When the base address and area are not specified, then zero and the current area are the defaults. If a `.setdp` directive is not issued the assembler defaults the direct page to the area "`_CODE`" at offset 0.

The assembler verifies that any local variable used in a direct variable reference is located in this area. Local variable and constant value direct access addresses are checked to be within the address range from 0 to 255.

External direct references are assumed by the assembler to be in the correct area and have valid offsets. The linker will check all direct page relocations to verify that they are within the correct area.

The 6809 microprocessor allows the selection of the direct page to be on any 256 byte boundary by loading the appropriate value into the dp register. Typically one would like to select the page boundary at link time, one method follows:

```
.area   DIRECT  (PAG)   ; define the direct page
.setdp
.
.
.
.area   PROGRAM
.
ldd     #DIRECT         ; load the direct page register
tfr     a,dp            ; for access to the direct page
```

At link time specify the base and global equates to locate the direct page:

```
-b DIRECT = 0x1000
-g DIRECT = 0x1000
```

Both the area address and offset value must be specified (area and variable names are independent). The linker will verify that the relocated direct page accesses are within the direct page.

The preceeding sequence could be repeated for multiple paged areas; however, an alternate method is to define a non-paged area and use the `.setdp` directive to specify the offset value:

```
.area   DIRECT          ; define non-paged area
.
.
.
```

```
.area   PROGRAM
.
.setdp  0,DIRECT        ; direct page area
ldd     #DIRECT         ; load the direct page register
tfr     a,dp            ; for access to the direct page
.
.
.setdp  0x100,DIRECT    ; direct page area
ldd     #DIRECT+0x100   ; load the direct page register
tfr     a,dp            ; for access to the direct page
```

The linker will verify that subsequent direct page references are in the specified area and offset address range. It is the programmers responsibility to load the `dp` register with the correct page segment corresponding to the `.setdp` base address specified.

For those cases where a single piece of code must access a defined data structure within a direct page and there are many pages, define a dummy direct page linked at address 0. This dummy page is used only to define the variable labels. Then load the `dp` register with the real base address, but do not use a `.setdp` directive. This method is equivalent to indexed addressing, where the `dp` register is the index register and the direct addressing is the offset.

1.5 Invoking ASxxxx

The ASxxxx assemblers are command line oriented. The PC assemblers are started with the appropriate option(s) and file(s) to assemble following the assembler name:

```
as6811 [-dqxgalosff] file1 [file2 file3 ...  file6]
```

The options are:

`d` decimal listing
`q` octal listing
`x` hex listing (default)

The listing radix affects the `.lst`, `.rel`, and `.sym` files.

`g` undefined symbols made global
`a` all user symbols made global

`l` create list output `file1.lst`
`o` create object output `file1.rel`
`s` create symbol output `file1.sym`

`p` disable listing pagination
`w` wide listing format for symbol table

`z` enable case sensitivity for symbols

Relocatable reference flagging:

`f` by ‘ in the listing file
`ff` by mode in the listing file

The file name for the .lst, .rel, and .sym files is the first file name specified in the command line. All output files are ascii text files which may be edited, copied, etc. The output files are the concatenation of all the input files, if files are to be assembled independently invoke the assembler for each file.

The .rel file contains a radix directive so that the linker will use the proper conversion for this file. Linked files may have different radices.

If the list (l) option is specified without the symbol table (s) option, the symbol table is placed at the end of the listing file.

1.6 Errors

The ASxxxx assemblers provide limited diagnostic error codes during the assembly process, these errors will be noted in the listing file and printed on the stderr device.

The assembler reports the errors on the stderr device as

```
?ASxxxx-Error-<*> in line nnn of filename
```

where * is the error code, nnn is the line number, and filename is the source/include file.

The errors are:

(.) This error is caused by an absolute direct assignment of the current location counter

```
. = expression (incorrect)
```

rather than the correct

```
. = . + expression
```

(a) Indicates a machine specific addressing or addressing mode error.

(b) Indicates a direct page boundary error.

(d) Indicates a direct page addressing error.

(i) Caused by an `.include` file error or an `.if`/`.endif` mismatch.

(m) Multiple definitions of the same label, multiple `.module` directives, or multiple conflicting attributes in an `.area` directive.

(o) Directive or mnemonic error or the use of the `.org` directive in a relocatable area.

(p) Phase error: label location changing between passes 2 and 3. Normally caused by having more than one level of forward referencing.

(q) Questionable syntax: missing or improper operators, terminators, or delimiters.

(r) Relocation error: logic operation attempted on a relocatable term, addition of two relocatable terms, subtraction of two relocatable terms not within the same programming area or external symbols.

(u) Undefined symbol encountered during assembly.

1.7 Listing File

The (l) option produces an ascii output listing file. Each page of output contains a four line header:

1. The ASxxxx program name and page number
2. Title from a .title directive (if any)
3. Subtitle from a .sbttl directive (if any)
4. Blank line

Each succeeding line contains five fields:

1. Error field (first three characters of line)
2. Current location counter
3. Generated code in byte format
4. Source text line number
5. Source text

The error field may contain up to 2 error flags indicating any errors encountered while assembling this line of source code.

The current location counter field displays the 16-bit program position. This field will be in the selected radix.

The generated code follows the program location. The listing radix determines the number of bytes that will be displayed in this field. Hexidecimal listing allows six bytes of data within the field, decimal and

octal allow four bytes within the field. If more than one field of data is generated from the assembly of a single line of source code, then the data field is repeated on successive lines.

The source text line number is printed in decimal and is followed by the source text.

Two special cases will disable the listing of a line of source text:

1. Source line with a `.page` directive is never listed.
2. Source line with a `.include` file directive is not listed unless the `.include` file cannot be opened.

Two data field options are available to flag those bytes which will be relocated by the linker. If the `-f` option is specified then each byte to be relocated will be preceeded by the ‘‘’ character. If the `-ff` option is specified then each byte to be relocated will be preceeded by one of the following characters:

1. `*` paged relocation
2. `u` low byte of unsigned word or unsigned byte
3. `v` high byte of unsigned word
4. `p` `PCR` low byte of word relocation or `PCR` byte
5. `q` `PCR` high byte of word relocation
6. `r` low byte relocation or byte relocation
7. `s` high byte relocation

1.8 Symbol Table File

The symbol table has two parts:

1. The alphabetically sorted list of symbols and/or labels defined or referenced in the source program.
2. A list of the program areas defined during assembly of the source program.

The sorted list of symbols and/or labels contains the following information:

1. Program area number (none if absolute value or external)
2. The symbol or label
3. Directly assigned symbol is denoted with an (=) sign
4. The value of a symbol, location of a label relative to the program area base address (=0), or a **** indicating the symbol or label is undefined.
5. The characters: G – global, R – relocatable, and X – external.

The list of program areas provides the correspondence between the program area numbers and the defined program areas, the size of the program areas, and the area flags (attributes).

1.9 Object File

The object file is an ascii file containing the information needed by the linker to bind multiple object modules into a complete loadable memory image. The object module contains the following designators:

```
[XDQ][HL]
        X       Hexidecimal radix
        D       Decimal radix
        Q       Octal radix

        H       Most significant byte first
        L       Least significant byte first

H       Header
M       Module
A       Area
S       Symbol
T       Object code
R       Relocation information
P       Paging information
```

Refer to the linker for a detailed description of each of the designators and the format of the information contained in the object file.

2 The Linker

2.1 ASLINK Relocating Linker

ASLINK is the companion linker for the ASxxxx assemblers.

The program ASLINK is a general relocating linker performing the following functions:

1. Bind multiple object modules into a single memory image
2. Resolve inter-module symbol references
3. Combine code belonging to the same area from multiple object files into a single contiguous memory region
4. Search and import object module libraries for undefined global variables
5. Perform byte and word program counter relative (`pc` or `pcr`) addressing calculations
6. Define absolute symbol values at link time
7. Define absolute area base address values at link time
8. Produce Intel Hex or Motorola S19 output file
9. Produce a map of the linked memory image
10. Produce an updated listing file with the relocated addresses and data

2.2 Invoking ASLINK

The linker may run in the command line mode or command file modes. The allowed startup linker commands are:

`-c`/`-f` command line / command file modes
`-p`/`-n` enable/disable echo `file.lnk` input to `stdout`

If command line mode is selected, all linker commands come from `stdin`, if the command file mode is selected the commands are input from the specified file (extension must be `.lnk`).

Most sytems require the initial options to be entered on the command line:

```
ASLINK -[cfpn]
```

Some systems may request the arguments after the linker is started at a system specific prompt:

```
ASLINK
argv:  -[cfpn]
```

After invoking the linker the valid options are:

1. `-i`/`-s` — Intel Hex (`file.ihx`) or Motorola S19 (`file.s19`) image output file.
2. `-z` — Specifies that symbol names are case sensitive.
3. `-m` — Generate a map file (`file.map`). This file contains a list of the symbols (by area) with absolute addresses, sizes of linked areas, and other linking information.

4. `-w` — Specifies that a wide listing format be used for the map file.
5. `-x`/`-d`/`-q` — Specifies the number radix for the map file (Hexidecimal, Decimal, or Octal).
6. `-u` — Generate an updated listing file (`file.rst`) derived from the relocated addresses and data from the linker
7. `fileN` — Files to be linked. Files may be on the same line as the above options or on a separate line(s) one file per line or multiple files separated by spaces or tabs.
8. `-b area = expression` (one definition per line) — This specifies an area base address where the expression may contain constants and/or defined symbols from the linked files.
9. `-g symbol = expression` (one definition per line) — This specifies the value for the symbol where the expression may contain constants and/or defined symbols from the linked files.
10. `-k library directory path` (one definition per line) — This specifies one possible path to an object library. More than one path is allowed.
11. `-l library file specification` (one definition per line) — This specifies a possible library file. More than one file is allowed.
12. `-e` — or null line, terminates input to the linker.

2.3 Library Path(s) and File(s)

The process of resolving undefined symbols after scanning the input object files includes the scanning of object module libraries. The linker will search through all combinations of the library path specifications (input by the `-k` option) and the library file specifications (input by the `-l` option) that lead to an existing library file. Each library file contains a list (one file per line) of modules included in this particular library. Each existing object module is scanned for a match to the undefined symbol. The first module containing the symbol is then linked with the previous modules to resolve the symbol definition. The library object modules are rescanned until no more symbols can be resolved. The scanning algorithm allows resolution

of back references. No errors are reported for non existent library files or object modules.

The library file specification may be formed in one of two ways:

1. If the library file contained an absolute path/file specification then this is the object module's path/file. (i.e., `C:\...`)
2. If the library file contains a relative path/file specification then the concatenation of the path and this file specification becomes the object module's path/file. (i.e., `\...`)

As an example, assume there exists a library file `termio.lib` in the `syslib` directory specifying the following object modules:

```
\6809\io_disk        first object module
d:\special\io_comm   second object module
```

and the following parameters were specified to the linker:

```
-k c:\iosystem\    the first path
-k c:\syslib\      the second path

-l termio          the first library file
-l io              the second library file (no such file)
```

The linker will attempt to use the following object modules to resolve any undefined symbols:

```
c:\syslib\6809\io_disk.rel      (concatenated path/file)
d:\special\io_comm.rel          (absolute path/file)
```

all other path(s)/file(s) don't exist. (No errors are reported for non existent path(s)/file(s).)

2.4 ASLINK Processing

The linker processes the files in the order they are presented. The first pass through the input files is used to define all program areas, the section area sizes, and symbols defined or referenced. Undefined symbols will initiate a search of any specified library file(s) and the importing of the module containing the symbol definition. After the first pass the `-b` (area base address) definitions, if any, are processed and the areas linked.

The area linking proceeds by first examining the area types `ABS`, `CON`, `REL`, `OVR` and `PAG`. Absolute areas (`ABS`) from separate object modules are always overlayed and have been assembled at a specific address, these are not normally relocated (if a `-b` command is used on an absolute area the area will be relocated). Relative areas (normally defined as `REL|CON`) have a base address of `0x0000` as read from the object files, the `-b` command specifies the beginning address of the area. All subsequent relative areas will be concatenated with proceeding relative areas. Where specific ordering is desired, the first linker input file should have the area definitions in the desired order. At the completion of the area linking all area addresses and lengths have been determined. The areas of type `PAG` are verified to be on a 256 byte boundary and that the length does not exceed 256 bytes. Any errors are noted on `stderr` and in the map file.

Next the global symbol definitions (`-g` option), if any, are processed. The symbol definitions have been delayed until this point because the absolute addresses of all internal symbols are known and can be used

in the expression calculations.

Before continuing with the linking process the symbol table is scanned to determine if any symbols have been referenced but not defined. Undefined symbols are listed on the `stderr` device. if a `.module` directive was included in the assembled file the module making the reference to this undefined variable will be printed.

Constants defined as global in more than one module will be flagged as multiple definitions if their values are not identical.

After the preceeding processes are complete the linker may output a map file (`-m` option). This file provides the following information:

1. Global symbol values and label absolute addresses
2. Defined areas and there lengths
3. Remaining undefined symbols
4. List of modules linked
5. List of library modules linked
6. List of `-b` and `-g` definitions

The final step of the linking process is performed during the second pass of the input files. As the `xxx.rel` files are read the code is relocated by substituting the physical addresses for the referenced symbols and areas and may be output in Intel or Motorola formats. The number of files linked and symbols defined/referenced is limited by the processor space available to build the area/symbol lists. If the `-u` option is specified then the listing files (`file.lst`) associated with the relocation files (`file.rel`) are scanned and used to create a new file (`file.rst`) which has all addresses and data relocated to their final values.

2.5 Linker Input Format

The linkers' input object file is an ascii file containing the information needed by the linker to bind multiple object modules into a complete loadable memory image.

The object module contains the following designators:

```
[XDQ][HL]
        X       Hexidecimal radix
        D       Decimal radix
        Q       Octal radix

        H       Most significant byte first
        L       Least significant byte first

H       Header
M       Module
A       Area
S       Symbol
T       Object code
R       Relocation information
P       Paging information
```

2.5.1 Object Module Format

The first line of an object module contains the [XDQ][HL] format specifier (i.e., XH indicates a hexadecimal file with most significant byte first) for the following designators.

2.5.2 Header Line

```
H aa areas gg global symbols
```

The header line specifies the number of areas (aa) and the number of global symbols (gg) defined or referenced in this object module segment.

2.5.3 Module Line

```
M name
```

The module line specifies the module name from which this header segment was assembled. The module line will not appear if the .module directive was not used in the source program.

2.5.4 Symbol Line

```
S string Defnnnn
```

or

```
S string Refnnnn
```

The symbol line defines (Def) or references (Ref) the symbol 'string' with the value nnnn. The defined value is relative to the current area base address. References to constants and external global symbols will always appear before the first area definition. References to external symbols will have a value of zero.

2.5.5 Area Line

```
A label size ss flags ff
```

The area line defines the area label, the size (ss) of the area in bytes, and the area flags (ff). The area flags specify the ABS, REL, CON, OVR, and PAG parameters:

```
OVR/CON  (0x04/0x00 i.e., bit position 2)

ABS/REL  (0x08/0x00 i.e., bit position 3)

PAG      (0x10 i.e., bit position 4)
```

2.5.6 T Line

```
T xx xx nn nn nn nn nn ...
```

The T line contains the assembled code output by the assembler with xx xx being the offset address from the current area base address and nn being the assembled instructions and data in byte format.

2.5.7 R Line

```
R 0 0 nn nn n1 n2 xx xx ...
```

The R line provides the relocation information to the linker. The nn nn value is the current area index, i.e., which area the current values were assembled. Relocation information is encoded in groups of 4 bytes:

1. n1 is the relocation mode and object format, for the adhoc extension modes refer to asxxxx.h or aslink.h
   (a) bit 0 word(0x00)/byte(0x01)
   (b) bit 1 relocatable area(0x00)/symbol(0x02)
   (c) bit 2 normal(0x00)/PC relative(0x04) relocation
   (d) bit 3 1-byte(0x00)/2-byte(0x08) object format for byte data
   (e) bit 4 signed(0x00)/unsigned(0x10) byte data
   (f) bit 5 normal(0x00)/page '0'(0x20) reference
   (g) bit 6 normal(0x00)/page 'nnn'(0x40) reference

(h) bit 7 LSB byte(0x00)/MSB byte(0x80) with 2-byte mode

2. n2 is a byte index into the corresponding (i.e., preceeding) T line data (i.e., a pointer to the data to be updated by the relocation). The T line data may be 1-byte or 2-byte byte data format or 2-byte word format.
3. xx xx is the area/symbol index for the area/symbol being referenced. the corresponding area/symbol is found in the header area/symbol lists.

The groups of 4 bytes are repeated for each item requiring relocation in the preceeding T line.

2.5.8 P Line

```
P 0 0 nn nn n1 n2 xx xx
```

The P line provides the paging information to the linker as specified by a .setdp directive. The format of the relocation information is identical to that of the R line. The corresponding T line has the following information:

```
T xx xx aa aa bb bb
```

Where aa aa is the area reference number which specifies the selected page area and bb bb is the base address of the page. bb bb will require relocation processing if the 'n1 n2 xx xx' is specified in the P line. The linker will verify that the base address is on a 256 byte boundary and that the page length of an area defined with the PAG type is not larger than 256 bytes.

The linker defaults any direct page references to the first area defined in the input REL file. All ASxxxx assemblers will specify the _CODE area first, making this the default page area.

2.6 Linker Error Messages

The linker provides detailed error messages allowing the programmer to quickly find the errant code. As the linker completes pass 1 over the input file(s) it reports any page boundary or page length errors as follows:

```
?ASlink-Warning-Paged Area PAGE0 Boundary Error
```

and/or

```
?ASlink-Warning-Paged Area PAGE0 Length Error
```

where PAGE0 is the paged area.

During Pass two the linker reads the T, R, and P lines performing the necessary relocations and outputting the absolute code. Various errors may be reported during this process The P line processing can produce only one possible error:

```
?ASlink-Warning-Page Definition Boundary Error
         file        module      pgarea      pgoffset
  PgDef  t6809l      t6809l      PAGE0       0001
```

The error message specifies the file and module where the .setdp direct was issued and indicates the page area and the page offset value determined after relocation.

The R line processing produces various errors:

```
?ASlink-Warning-Byte PCR relocation error for symbol  bra2
         file        module      area        offset
  Refby  t6809l      t6809l      TEST        00FE
  Defin  tconst      tconst      .   .ABS.    0080

?ASlink-Warning-Unsigned Byte error for symbol  two56
         file        module      area        offset
  Refby  t6800l      t6800l      DIRECT      0015
  Defin  tconst      tconst      .   .ABS.    0100

?ASlink-Warning-Page0 relocation error for symbol  ltwo56
         file        module      area        offset
  Refby  t6800l      t6800l      DIRECT      000D
  Defin  tconst      tconst      DIRECT      0100

?ASlink-Warning-Page Mode relocation error for symbol  two56
         file        module      area        offset
  Refby  t6809l      t6809l      DIRECT      0005
  Defin  tconst      tconst      .   .ABS.    0100
```

```
?ASlink-Warning-Page Mode relocation error
         file        module      area        offset
  Refby  t           Pagetest    PROGRAM     0006
  Defin  t           Pagetest    DIRECT      0100
```

These error messages specify the file, module, area, and offset within the area of the code referencing (`Refby`) and defining (`Defin`) the symbol. If the symbol is defined in the same module as the reference the linker is unable to report the symbol name. The assembler listing file(s) should be examined at the offset from the specified area to located the offending code.

The errors are:

1. The byte `PCR` error is caused by exceeding the `pc` relative byte branch range.
2. The Unsigned byte error indicates an indexing value was negative or larger than 255.
3. The `Page0` error is generated if the direct page variable is not in the `page0` range of 0 to 255.
4. The page mode error is generated if the direct variable is not within the current direct page (6809).

2.7 Intel Hex Output Format

Record Mark Field — This field signifies the start of a record, and consists of an ascii colon (`:`).

Record Length Field

This field consists of two ascii characters which indicate the number of data bytes in this record. The characters are the result of converting the number of bytes in binary to two ascii characters, high digit first. An `End of File` record contains two ascii zeros in this field.

Load Address Field

This field consists of the four ascii characters which result from converting the the binary value of the address in which to begin loading this record. The order is as follows:

- High digit of high byte of address.
- Low digit of high byte of address.
- High digit of low byte of address.
- Low digit of low byte of address.

In an `End of File` record this field consists of either four ascii zeros or the program entry address. Currently the entry address option is not supported.

Record Type Field

This field identifies the record type, which is either 0 for data records or 1 for an `End of File` record. It consists of two ascii characters, with the high digit of the record type first, followed by the low digit of the record type.

Data Field

This field consists of the actual data, converted to two ascii characters, high digit first. There are no data bytes in the `End of File` record.

Checksum Field

The checksum field is the 8 bit binary sum of the record length field, the load address field, the record type field, and the data field. This sum is then negated (2's complement) and converted to two ascii characters, high digit first.

2.8 Motorla S1-S9 Output Format

Record Type Field

This field signifies the start of a record and identifies the record type as follows:

- Ascii `S1` – Data Record
- Ascii `S9` – `End of File` Record

Record Length Field

This field specifies the record length which includes the address, data, and checksum fields. The 8 bit record length value is converted to two ascii characters, high digit first.

Load Address Field

This field consists of the four ascii characters which result from converting the the binary value of the address in which to begin loading this record. The order is as follows:

- High digit of high byte of address.
- Low digit of high byte of address.
- High digit of low byte of address.
- Low digit of low byte of address.

In an `End of File` record this field consists of either four ascii zeros or the program entry address. Currently the entry address option is not supported.

Data Field

This field consists of the actual data, converted to two ascii characters, high digit first. There are no data bytes in the `End of File` record.

Checksum Field

The checksum field is the 8 bit binary sum of the record length field, the load address field, and the data field. This sum is then complemented (1's complement) and converted to two ascii characters, high digit first.

3 AS6811 Assembler

3.1 68HC11 Register Set

The following is a list of the 68HC11 registers used by AS6811:

`a`,`b`	8-bit accumulators
`d`	16-bit accumulator `<a:b>`
`x`,`y`	index registers

3.2 68HC11 Instruction Set

The following tables list all 68HC11 mnemonics recognized by the AS6811 assembler. The designation `[]` refers to a required addressing mode argument. The following list specifies the format for each addressing mode supported by AS6811:

`#data`	immediate data byte or word data
`*dir`	direct page addressing (see `.setdp` directive) 0 <= `dir` <= 255
`,x`	register indirect addressing zero offset
`offset,x`	register indirect addressing 0 <= `offset` <= 255
`ext`	extended addressing
`label`	branch label

The terms `data`, `dir`, `offset`, and `ext` may all be expressions.

Note that not all addressing modes are valid with every instruction, refer to the 68HC11 technical data for valid modes.

3.2.1 Inherent Instructions

`aba` `abx` `aby` `cba`

```
clc     cli     clv     daa
des     dex     dey     fdiv
idiv    ins     inx     iny
mul     nop     rti     rts
sba     sec     sei     sev
stop    swi     tab     tap
tba     tpa     tsx     txs
wai     xgdx    xgdy

psha    pshb    psh a   psh b
pshx    pshy    psh x   psh y

pula    pulb    pul a   pul b
pulx    puly    pul x   pul y
```

3.2.2 Branch Instructions

```
bra  label      brn  label      bhi  label      bls  label
bcc  label      bhs  label      bcs  label      blo  label
bne  label      beq  label      bvc  label      bvs  label
bpl  label      bmi  label      bge  label      blt  label
bgt  label      ble  label      bsr  label
```

3.2.3 Single Operand Instructions

```
asla            aslb            asld
asl a           asl b           asl d           asl     []

asra            asrb            asr a           asr b
asr     []

clra            clrb            clr a           clr b
clr     label

coma            comb            com a           com b
com     []

deca            decb            dec a           dec b
dec     []

inca            incb            inc a           inc b
inc     []

lsla            lslb            lsld
lsl a           lsl b           lsl d            lsl     []

lsra            lsrb            lsrd
lsr a           lsr b           lsr d            lsr     []
```

```
nega            negb            neg a           neg b
neg     []

rola            rolb            rol a           rol b
rol     []

rora            rorb            ror a           ror b
ror     []

tsta            tstb            tst a           tst b
tst     []
```

3.2.4 Double Operand Instructions

```
adca    []      adcb    []      adc a   []      adc b   []

adda    []      addb    []      addd    []
add a   []      add b   []      add d   []

anda    []      andb    []      and a   []      and b   []

bita    []      bitb    []      bit a   []      bit b   []

cmpa    []      cmpb    []      cmp a   []      cmp b   []
```

```
eora    []      eorb    []      eor a   []      eor b   []

ldaa    []      ldab    []      lda a   []      lda b   []

oraa    []      orab    []      ora a   []      ora b   []

sbca    []      sbcb    []      sbc a   []      sbc b   []

staa    []      stab    []      sta a   []      sta b   []

suba    []      subb    []      subd    []
sub a   []      sub b   []      sub d   []
```

3.2.5 Bit Manupulation Instructions

```
bclr    [],#data                bset    [],#data

brclr   [],#data,label          brset   [],#data,label
```

3.2.6 Jump and Jump to Subroutine Instructions

```
jmp     []              jsr     []
```

3.2.7 Long Register Instructions

cpx	[]	cpy	[]				
ldd	[]	lds	[]	ldx	[]	ldy	[]
std	[]	sts	[]	stx	[]	sty	[]