SOFTWARE DEVELOPMENT SYSTEM

ASSEMBLER/LINKER/LIBRARIAN

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 1 Introduction

This manual provides all the information needed to develop assembly language programs for the W65xxx series of microprocessors using the WDC Software Development System. The WDC software development system consists of a two-pass macro assembler, an object module linker, and an object module librarian. There are two WDC assemblers called WDC02AS and WDC816AS. They are identical in function with two exceptions. First, the default instruction set for the WDC02AS assembler is the 65C02 while the default instruction set for the WDC816AS assembler is the 65C816. Second, the WDC02AS assembler checks for the environment variable WDC_INC_6502 while the WDC816AS assembler checks for WDC_INC_65816. Throughout this manual, when the assembler is referenced, the name WDCxAS will be used.

Assembler

The WDC assembler, WDCxAS, translates assembly language source files into object files. The assembler supports the full 6502 and 65816 instruction sets including alternate mnemonics for a number of the instructions. Twenty-four different addressing modes are supported. Subsets of the full instruction set for the 6502 and 65C02 can be selected. The extra instructions of the R65C02 can also be selected. Assembler directives control the organization of source files into modules for the creation of libraries. Other directives allow the creation and nesting of up to 250 named sections of code or data as well as 5 pre-defined sections. Sections can be ``org"ed at an absolute address or can be located by the linker. Symbols can be made private to a source file or public so they can be referenced by other source files. Common definition files can be referenced by any source file. Additional source files can be appended to any source file.

Assembly language source statement parsing can be tailored by the setting of the default number base, tolerance of spaces in operands, and the size of index registers and accumulator. A full range of directives is available for the generation of data. Block fills of memory, ASCII strings, and byte, word or long size constant data can be generated. There are also directives for controlling how ASCII strings are converted to bytes, a date directive for generating the ascii byte data equivalent of the current date, as well as the ability to reserve blocks of memory with no initial value.

The assembler also provides a fast macro capability with argument passing. Macros may be nested and may recurse. Special label handling is available for recursive macros. A full range of conditional directives make macros extremely powerful. The assembler can optionally produce a listing file. This file can be tailored to list only those sections of code desired. The page size, length, heading and subheading are all modifiable. Listing of expanded macros and false conditionals are also controlled by assembler commands. All in all, the assembler provides all the facilities needed for modern assembly language program development.

Linker

The assembler does not generate output files directly. Instead, object files are created that can be combined together with other object files and possibly libraries to produce the final output program. The program that combines object modules and produces the final hex output is the linker, WDCLN. The linker acts as an organizer. It reads all the object files determining where each will end up in memory or in ROM. The linker then patches any references that occur from one file to another with the proper address. Finally, the linker produces the hex output file and symbol table in the formats requested.

Librarian

The librarian, WDCLIB, collects object files together into a single library file. The librarian places a special dictionary of all the functions defined within the library at the front of the library. The linker can use this dictionary to find the functions it needs in the library very quickly. Only the functions needed by the current program are copied from the library into the output file.

Manual organization

There are a lot of details in dealing with an assembly language development system. As a result, this manual is broken into two basic parts. The first part provides a general overview of the development system and how to use it. In particular, it covers the process of creating assembly language programs, the program structure, the syntax of assembly language statements and how to write and use macros. The second part is more of a reference manual. It provides short descriptions of each of the programs with their options detailed, a list of all the assembler mnemonics and addressing modes, and all of the assembler directives grouped by functionality. Numerous examples are present throughout the text.

CHAPTER 2 Files

This CHAPTER discusses the general process of creating assembly language programs. It covers the whole development cycle and provides an overview of how the different pieces fit together.

Source files

All assembly language programs begin with the source file. The source file is a text file, which contains assembly language statements. Each statement contains a processor opcode, an assembler directive, a section directive or a macro name. Assembly language files are created using a text editor. Source files usually have an extension of *.ASM*. The source to an entire program does not need to all be in one file. There are several reasons for dividing a program into different source files. One reason involves the use of macros.

Macro files

Macros are sequences of assembly language statements that are defined and that can replace a single statement in the source. Macros allow commonly used sequences of instructions to be defined once and a name associated with that definition. Then, whenever the name is used, the name is replaced with the statements of the macro definition. Through the use of macro arguments and conditional expressions, macros can be a very powerful tool for assembly language development. As multiple programs are developed, a number of useful macros may be produced. Instead of placing these macros in the source file of each program, it is easier to place the macros in a file of their own. Then this file can be referenced by each program source file using the assembler **INCLUDE** directive.

Object modules and libraries

Modular programming is a second reason for having more than one source file. Different parts of the program can be developed and tested independently. Then, the individual parts can be combined together by the linker. In addition, useful subroutine functions can be created, assembled and combined together into a library. A librarian program creates libraries from object files. The advantage of having a library is that only the functions that are used by a program are copied from the library. It is a convenient way to organize and access common functions. The assembler translates each source file into an *object* file. An object file is a binary file which contains the translation of each assembly language statement into it's binary equivalent. In addition, the object file contains a list of the symbols defined in the source file as well as those symbols referenced within the source file that aren't defined there.

Output files

The linker takes multiple object modules and combines them together to produce a single output file. Some of the object modules may come from a library file. The output file produced by the linker is either in binary or hex format. A number of hex formats are supported. An option to the linker selects the desired hex format. This hex file can be used to program ROMs or can be downloaded into an emulator for testing.

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 3 Program Structure

Each program has a definite structure. All programs consist of two basic quantities *CODE* and *DATA*. CODE consists of all the instructions that control the processor and tell it what to do. DATA is the part of the program where information is stored.

Modules

As previously discussed, each program consists of one or more source files. Each source file usually contains both CODE and DATA statements. The assembler supports a concept called *module*. A module is a set of assembly language statements that can be assembled independently. Most source files are considered a single module by the assembler. Using the **MODULE** directive in the assembler, multiple modules can be contained in a single source file. As the assembler encounters each module, it treats each module as though it had occurred in its own separate file. The only real use and advantage of creating multiple modules is in creating libraries. If multiple functions were defined in a single source file then calling just one of the functions would force all of the functions to be included in the output file. Since the module is the smallest unit that the linker will work with, placing each function in its own module allows the linker to only select the functions that are referenced.

Sections

Whereas modules provide a means of organizing source code, *sections* are used to organize where the results ultimately are placed in memory. The easiest way to understand sections is to consider program code and program data. A typical program consists of a number of processor opcodes called the code and some amount and type of information called data. Since the processor is not smart enough to be able to distinguish code from data, the two cannot be intermixed at will. As a result most programs tend to contain a single block of program code and a single block of data. Sections are a way for the programmer to indicate whether the output should be considered code or data. A program is organized into sections. There are a number of pre-defined sections. Two of these are CODE and DATA. When a source file is assembled, the assembler assumes that the initial section is a CODE section. At any time, a different section can be activated by using the name of the section as a directive. When a different section is activated, the previous section is pushed onto a stack. When a section is ended with the **ENDS** directive, the section stack is popped and the previous section becomes active again.. When the program is linked all of the pieces of each section are joined together.
Note: **Sections can be nested up to 500 deep**
Note: **The name of a section is truncated to 8 characters on the Linker display screen.**
Thus, the input source file looks like:

```
CODE
code1 statements
        DATA
        data1 statements
        ENDS
code2 statements
        DATA
        data2 statements
                CODE
                code3 statements
                ENDS
        data3 statements
        ENDS
END
```


When the output file is assembled, the program will look like:

```
CODE
code1 statements
code2 statements
code3 statements
DATA
data1 statements
data2 statements
data3 statements
END
```

Thus, sections provide a convenient method of grouping the source together of both code and data while still maintaining the distinction between them. A typical example is a subroutine that maintains its own work variables. Using sections it is easy to keep the data with the code.

For example:

```
        DATA
TMP     DB          0
        ENDS

SWAP:   LDA         ARRAY,X     ;get value
        STA         TMP         ;save it
        LDA         ARRAY,Y     ;get other value
        STA         ARRAY,X     ;copy it
        LDA         TMP         ;get saved value
        STA         ARRAY,Y     ;and copy that
        RTS                     ;all done, return
```

Pre-defined sections

The assembler has five pre-defined sections whose names and descriptions are:

PAGE0 This section is reference only and is used for creating labels that refer to the direct page. For the 6502, address $00-$FF. For the 65816, address $00:0000- $00:FFFF.
CODE This is the main program section.
KDATA This is a special data section for constant initialized data that is never modified. For example, a lookup table of constants.
DATA This is initialized data that will be modified.
UDATA This is data that is not initialized.

Three different data sections have pre-defined characteristics to promote efficient use of ROM and RAM memory space. In addition, up to 250 additional sections can be created and named by the programmer using the **SECTION** directive.

Absolute versus Relative

Sections can be either absolute or relocatable. An absolute section starts at a fixed address as specified by an **ORG** statement in the source code. When the assembler sees an **ORG** statement, it marks the section as absolute and as it assembles each statement, it treats any labels defined in the section as being exactly at the absolute location specified. A relocatable section works a little different. When the assembler assembles each statement, it treats each label as being relative to the beginning of the section. Nothing is considered absolute. Then, after the linker collects all the pieces of each section, the entire section can be located at an absolute address by the linker. The linker will then adjust all the references to relocatable labels turning them into absolute.

All sections begin as relocatable sections unless created with the **SECTION** directive followed by either the **OFFSET** or **INDIRECT** options. A section also becomes absolute if an **ORG** directive occurs within the section.

Section location

If a program is composed of several different pieces of code and/or data that must be located in various locations, then absolute sections and the **ORG** directive are definitely the way to go. For example, to place the page number in the first two bytes of the first three pages of bank zero, consider the following source fragment:

```
DATA
ORG     $0000
WORD    $0000

ORG     $0100
WORD    $0001

ORG     $0200
WORD    $0002
ENDS
```

On the other hand, if all that is needed is to put the code one place and the data a different place, then relocatable sections are worthwhile. Since relocatable sections do not start at a fixed address, the linker is used to place these sections. Using the linker it is possible to place each of the sections at a particular address. Alternatively, the location of the code section can be placed and the linker will automatically place any remaining sections one after the other. In this case, the only option to the linker is the **–C** option followed by the address where the code is to be placed. When the starting address of a section is not specified, the linker places it immediately after the preceding section. The first four sections are always CODE, KDATA, DATA, and UDATA in that order. Any user-defined sections follow UDATA and occur in the order in which the linker encounters them.

Copying data

In some applications, the program resides in ROM. In this case, the initialized data usually resides in ROM as well. However, what is desired is to copy the data to RAM and have the code in ROM access the data at its RAM address, not the ROM address. This is easily accomplished using linker options to set the RAM and ROM address of the DATA section. For example:

```
WDCLN -C8000 -D1000,   MYPROG.OBJ
```

will place the CODE section starting at 8000 hex. This is both the ROM address and the address at which the code is expected to run. The second argument will relocate the DATA section and all references to the DATA section to hex 1000. However, since an empty ROM address is specified in the argument, the ROM address of the DATA section will be immediately following the KDATA section. If the KDATA section is empty, the DATA section will immediately follow the end of the CODE section. In this case, when the program starts up, the DATA section will need to be copied from ROM to RAM. The included example startup file *STARTUP.ASM* shows one method of achieving this. This code also sets any unutilized data to zero, sets up the stack pointer and the Data Bank Register.

Startup.ASM

Included on the distribution disk is a sample assembly language source file called *STARTUP.ASM*. This file contains code that performs several important functions related to program startup. It also contains a section which defines the standard interrupt and reset vectors. This file can be customized to match the application that you are developing. We will examine the 816 version of this file section by section.

```
STACK       EQU         $F000           ;CHANGE THIS FOR YOUR SYSTEM

STARTUP     SECTION     OFFSET $FF80
START:
            CLC                         ;clear carry
            XCE                         ;clear emulation
            REP         #$30            ;16 bit registers
            LONGI       ON
            LONGA       ON
            LDA         #STACK          ;get the stack address
            TCS                         ;and set the stack to it
```

The first section does three things that are necessary before the remaining sections can run properly. First, it defines a new section called *STARTUP* which will be located at the end of bank 0 at location $FF80. Next, we assume that we arrive here from the RESET vector so we will be in emulation mode. Thus, the first thing we do is switch to native mode and make sure that the registers are sixteen-bits wide and that the assembler knows it as well. Finally, we set the up the hardware stack pointer. The **STACK** equate should be changed to reflect where you wish the top of the stack to be in your system.

```
            SEP         #$20            ;8 bit accum
            LONGA       OFF
            LDA         #^_BEG_DATA     ;get bank of data
            PHA
            PLB                         ;set data bank register
            REP         #$20            ;back to 16 bit mode
            LONGA       ON
```

This piece of code sets the Data Bank register by pushing the bank part of the DATA section and then popping it into the register. The *_ROM_BEG_DATA* and *_BEG_DATA* symbols are symbols automatically created by the linker. For each section, it creates three symbols, *_ROM_BEG_secname*, *_BEG_secname* and *_END_secname*, which correspond to the rom location and the execution beginning and end of the section. These will be used more in the next two sections of code.

```
            LDA         #_END_DATA-_BEG_DATA        ;number of bytes to copy
            BEQ         SKIP                        ;if none, just skip
            DEC         A                           ;less one for MVN instruction
            LDX         #<_ROM_BEG_DATA             ;get source into X
            LDY         #<_BEG_DATA                 ;get dest into Y
            MVN         #^_ROM_BEG_DATA,#^_BEG_DATA ;copy bytes
SKIP:
```

Next, we copy the DATA from ROM to RAM. This section assumes that the DATA has been linked to reside in RAM, but is physically in the ROM at some location specified by *_ROM_BEG_secname*. If the DATA does not need to be copied, this section can be deleted. First, we calculate the size of the DATA section by subtracting the end of the section from the beginning. If the size is zero, we skip ahead. If the size is non-zero, we decrement it, load the X and Y registers with the low sixteen bits of the ROM and RAM addresses. Finally, we copy the data using the MVN instruction with the bank parts of the ROM and RAM addresses.


```
        LDX   #_END_UDATA-_BEG_UDATA    ;get number of bytes to clear
        BEQ   DONE                      ;nothing to do
        LDA   #0                        ;get a zero for storing
        SEP   #$20                      ;do byte at a time
        LDY   #_BEG_UDATA               ;get beginning of zeros

LOOP    STA   |0,Y                      ;clear memory
        INY                             ;bump pointer
        DEX                             ;decrement count
        BNE   LOOP                      ;continue till done
        REP   #$20                      ;16 bit memory reg
DONE:
```

Next, we need to fill the un-initialized data area, UDATA, with zeroes. First, we calculate the size of the UDATA section and if it is zero, skip ahead. Then, we get zero into the accumulator and make it eight bits wide. Next, we put the beginning of the UDATA section into the Y register. Finally, we loop through all of the UDATA section storing the zero in the accumulator. When the loop finishes, we restore the accumulator to sixteen bits.

```
        XREF  MYSTART           ;change MYSTART to yours
        JMP   >MYSTART          ;long jump in case not bank 0

        XREF  _ROM_BEG_DATA
        XREF  _BEG_DATA
        XREF  _END_DATA
        XREF  _BEG_UDATA
        XREF  _END_UDATA
```

This last section of code performs a long absolute jump to the start of the actual program. This allows the program to reside anywhere in the processors address space. The XREF directive tells the assembler that the *MYSTART* symbol is defined in another file and will be filled in by the linker. Change *MYSTART* to be the name of the entry point of your assembly language program. It is also necessary to define the entry point to be public in the file that defines it using XDEF. The last five directives declare the linker special symbols external so that the linker will know to fill them in.

```
ORG         $FFE4

N_COP       DW    0
N_BRK       DW    0
N_ABORT     DW    0
N_NMI       DW    0
N_RSRVD     DW    0
N_IRQ       DW    0
            DS    4
E_COP       DW    0
E_RSRVD     DW    0
E_ABORT     DW    0
E_NMI       DW    0
E_RESET     DW    START
E_IRQ       DW    0
            ENDS
            END
```

The last section implements the reset and interrupt vectors. The **ORG** directive makes sure that they are in the right location. The only vector that is filled in is the RESET vector which points to the *START* label which is at the beginning of STARTUP.ASM. To use any of the remaining vectors simply replace the `0' with the label of the corresponding interrupt

routine. Note that the vectors are only sixteen-bits wide which limits them to an address in Bank zero. To use an interrupt routine that is not in Bank zero, add the following lines before the **ORG** directive for each vector you wish to use:

```
JMP0    JMP     >FUNC0
```

Then, place the label *JMP0* in the vector table. The basic idea is to vector into Bank zero to a long absolute jump to the interrupt handler.

Nintendo development

This section briefly describes those features of the development system which have been provided to aid in development for the Super Nintendo Entertainment System (SNES). The linker provides three options which are specifically designed to enhance SNES development. The **-HN** option produces ISX binary format that can be used with the Nintendo debugger. Full symbol support is provided along with LONGA/LONGI disassembly support. The **-MN**, **-MN80** and **-MN21** options are provided primarily for C language programmers who don't wish to specify the location of their routines explicitly. When the **-MN** option is specified, the Nintendo memory map is used when creating the final output program. The initial code segment starts at $00:8000 and each succeeding bank code segment starts at $8000 as well. The linker first places absolute sections and sections which have a ROM org address specified as a link option. Then it attempts to place code modules in any holes, expanding to new banks as needed. The **-MN80** option is exactly the same, but uses the fast ROM addresses starting with bank $80. The **-MN21** option operates similarly, but starts at $C0:0000 and starts at $0000 of each additional bank. Note that for C programs, they must have been compiled with Large Code when using these linker options.

CHAPTER 4 Statement Syntax

All lines other than comment lines and blank lines have the following format:

LABEL OPERATION OPERAND

Labels are optional on most statements, required on a few and illegal on the remainder. Operands are required or illegal on most statements and optional on a few.

Comments

A comment normally appears after a semicolon. If an entire line is to be a comment, an asterisk may be used instead. Comments also may appear after the operand field of an instruction, macro call or directive. By default, spaces are not allowed in the operand field, so a space before the comment will be sufficient. However, if the **SPACES ON** directive has been given, a semicolon is required to remove the ambiguity.

Labels

Labels give the programmer the ability to give a name to a location, value or macro that occurs in the assembly language program. When a label is defined it must be the first thing on that line. It may be preceded by spaces or tabs only if it is followed by a colon, otherwise it must begin with the first character of the line. Labels can be up to 64 characters long. A label must begin with a letter or either of the characters `_' or `~'. The rest of the label must contain letters, numbers, or the characters `_' or `~'.

Some examples of labels are:

```
1                   9       (columns)
lab1
        lab2:
__foo
~~main
```

By default labels are only visible to the module or file in which they are defined. If the a label is made global using one of the **XDEF**, **GLOBAL**, or **PUBLIC** directives, then it is visible to any other module.

There are also temporary labels that have an even smaller range of visibility than normal labels. Temporary labels are only visible from one normal label to the next normal label. Temporary labels begin or end with a ? and may contain any of the characters contained in a normal label. Temporary labels that begin with a ? are different from those that end with one. The ? character can be changed by using the **LLCHAR** directive. Labels are optional on all processor instructions and macro calls and on most assembler directives. A label may also appear on a line by itself.

WARNING: The following are reserved labels: A, X and Y.

Operation

There are several different types of operations that can occur in an assembly language statement. All operation opcodes are case independent. In addition, except for processor instructions, all directives may be preceded by a `.' to distinguish them from processor opcodes.

Processor Instructions

The full list of processor instruction mnemonics is presented in CHAPTER 11. The subset of valid processor mnemonics is dependent on the last **CHIP** directive executed. The WDC816AS assembler defaults to **CHIP 65816** and the WDC02AS assembler defaults to **CHIP 65C02**, however other processors in the same family can be selected. Each instruction has one or more allowable addressing modes. These are described in the data sheets that accompany each processor. The full set of addressing modes supported by the W65C816 processor is described in CHAPTER 11.

Assembler Directives

There are a wide and varied assortment of assembler directives which control various aspects of the assembly process. CHAPTER 11 provides detailed information on each assembler directive supported. The assembler directives are grouped into six main areas:

- File and Symbol Control
- Parsing Control
- Data Definition Control
- Macro Control
- Conditional Control
- Listing Control

Section Directives

Section names can be used as an assembler directive instructing the assembler to place subsequent statements in the named section. There are five predefined section names: PAGE0, CODE, KDATA, DATA, and UDATA. Up to 250 additional section names can be created.

Macro Calls

A macro, once it has been defined, can be used like any other assembler directive. Macro calls may have labels and may or may not have arguments specified in the operand field. See CHAPTER 5 on Macros and Conditionals for more details.

Operands

Operators

The assembler supports a number of operators. Each operator has a precedence associated with it. Operators with the same precedence are evaluated left to right. Parentheses can be used to group expressions and change the operator precedence. For example, the expression:

```
a+b*c
```

is interpreted as:

```
a+(b*c)
```

because the `*' has a higher precedence than the `+'. Parentheses can be used to re-order the expression to achieve the desired result:

```
(a+b)*c
```

Unary Operators

Unary operators are followed by a single expression.

+ EXPR	specifies a positive expression
- EXPR	negates the value of EXPR
.NOT. EXPR	bit-wise complement of EXPR
/ EXPR	

Binary Operators

L ** R	raise L to the R power
L * R	multiply L times R
L / R	divide L by R
L .MOD. R	get the remainder from dividing L by R
L >> R	logically shift L to the right R times
L .SHR. R	
L << R	shift L to the left R times
L .SHL. R	
L R	add L to R
L - R	subtract R from L
L & R	do a bit-wise AND of L and R
L .AND. R	
L | R	do a bit-wise OR of L and R
L .OR. R	
L ^ R	do a bit-wise XOR of L and R
L .XOR. R	

Comparison Operators

L = R	left expression equal to right
L .EQ. R	
L > R	left expression greater than right
L .GT. R	
L < R	left expression less than right
L .LT. R	
L .UGT. R	left expression unsigned greater than right
L .ULT. R	left expression unsigned less than right

Operator Precedence Table

Unary operators have the highest precedence, comparison operators have the lowest. In the following table, operators on the same line have the same precedence. Each succeeding line has lower precedence than the preceding line.

Unary operators (+, -, \, .NOT.)
**
*, /, .MOD., >>, .SHR., <<, .SHL.
+, -
&, .AND.
|, .OR, ^, .XOR.
Comparison operators (=, .EQ., >, .GT., <, .LT., .UGT., .ULT.)

Special symbols to help calculate Numbers and Addresses (**ADDRESSING MODE SYMBOLS**)

Symbol	Name	Use
|	Pipe	Absolute Address
!	Exclamation	Absolute Address
<	Less than	Lower 16 bits only
>	Greater than	8 bit shift to the right
#	Pound	Immediate
^	Caret	Upper 16 bits only
*	Star	Program Counter at the next location
$	Dollar	Program Counter current hex value
#( ).low.		Lower 16 bits only
#( ).high.		Upper 16 bits only
>>		16bit shift to right (used to target the bank, load bank address)
#“x”	(pound double quote)	Use ASCII character for Immediate.
#‘@’	(pound single quote)	Use ASCII character for Immediate. If LongA is on then # ‘23’ is valid.

Numbers

Numbers can be specified several different ways using different bases. The default base is decimal. The base can be changed by using the **RADIX** directive. To indicate a number that is not in the default base, a letter is appended to the number indicating the base to use.

The following table lists the prefixes and the corresponding base:

$	hexadecimal
%	binary

The following table lists the suffixes and the corresponding base:

B	binary
O or Q	octal
D	decimal
H	hexadecimal

Addresses

Addresses can be specified using a number as defined in the preceding section. Alternatively, an address can be specified as a bank number followed by a colon followed by an address.

For example:

```
LDA    2:30H
LDA    $20030
```

produce the same result.

When the assembler is parsing addresses, it starts by assuming that an address is ABSOLUTE. If the address is greater than FFFFH, then it assumes that it is a long absolute address. It is possible to directly specify how an address should be interpreted by the assembler. This is especially important when using symbolic names which are defined externally.
To specify an address that is a direct page address, the `<' character precedes the address or expression. For example, consider the following two statements:

```
LDA    0
LDA    <0
```

The first statement is interpreted as absolute location 0 in the bank specified by the Data Bank Register. The second statement will load from location 0 in the direct page. To force an address to be an absolute address, the `|' character or the `!' character precedes the address or expression. If an expression value is less than 10000H, then this prefix is redundant. Finally, to force a long absolute address, the `>' character should precede the address or expression.

For example:

```
LDA   |1:0000H        ;load from 0 absolute
LDA   >0              ;load from 0 in bank 0
```

In the first example, the bank value is ignored.
The address prefix operators must be used at the beginning of the expression.

Immediate Operands

When an instruction uses the IMMEDIATE addressing mode, the operand is preceded by a `#' character. The value of the expression following the `#' character is truncated to either eight or sixteen bits depending on the settings of the **CHIP**, **LONGA** and **LONGI**.

The `#' character may be followed by a second character which causes the expression to be shifted right before truncating. The `<' character causes no shift to occur, the `>' character causes an eight bit shift, and the `^' character causes a sixteen bit shift. Thus, these characters allow access to the low byte, the high byte and finally the bank part respectively.

For example:

```
#01020304H          04H     0304H
#<1020304H          04H     0304H
#>1020304H          03H     0203H
#^1020304H          02H     0102H
#(1020304H).low.    04H     0304H
#(1020304H).high.   03H     0102H
```

shows the effect of these prefixes with an eight-bit and sixteen-bit expression.

The expressions `#( ).low.' and `#( ).high.' can be used as well. The **DBREG** and **DPAGE** directives can be used to control the generation of addresses for absolute symbols. The **DBREG** is used to generate an absolute address if the symbol is located in the Data Bank specified in the **DBREG** directive. Otherwise, long absolute addressing will be used. Similarly, the **DPAGE** directive can be used to indicate the run-time value of the Direct Page register to allow the assembler to optimize generation of Direct Page address expressions. For ASCII characters, use LDA #“!”.

Character Constants and Strings

Character constants are delimited by quote characters. Special combinations of two character sequences may be used or certain control characters if enabled by the **TWOCHAR** directive. Character strings are delimited by either single or double quote characters.

Program Counter

The characters `$' or `*' can be used in an expression to represent the program counter at the beginning of the current instruction.

For example:

```
LAB   EQU   *
      BRA   $+10
```

Assembler Addressing Modes

8-bit operations less than $100 (Page zero)

Normal direct page addressing:
LDA $43 ;Load Accumulator from: bank zero: direct page: $43

Force absolute addressing: zero page in data bank:
LDA !$23 ;Load Accumulator from: data bank: $0023

Force long addressing: zero page in data bank zero:
LDA >$33 ;Load Accumulator from: $00:0033

16-bit Operations from $100 thru $FFFF

Normal absolute addressing:
LDA $5643 ;Load Accumulator : data bank: $5643

Force direct page addressing:
LDA <$5633 ;Load Accumulator from: bank zero: direct page: $33

Force long addressing:
LDA >$5633 ;Load Accumulator from: $00:5633

24-bit Operations over $FFFF

Normal long addressing:
LDA $456733 ;Load Accumulator from: $45:6733

Force absolute addressing::
LDA !$567833 ;Load Accumulator from: data bank: $7833

Force direct page addressing:
LDA <$567833 ;Load Accumulator from: bank zero: direct page: $33

CHAPTER 5 Macros and Conditionals

This CHAPTER describes how to write and use macros and conditionals.

Macros

Macros make complicated or repeated sets of assembly language instructions much simpler to use. Conditionals within macros allow a single macro to be useful in multiple situations. The use of macros greatly enhances the readability of assembly language source files.

Macro Definition

Macros must be defined before they are used. Each macro definition specifies the name of the macro and the names of the arguments to the macro if any. The syntax of the definition is:

```
LABEL       MACRO       [ARG1,ARG2,...]
```

where the *LABEL* specifies the name of the macro and the arguments are optional.

For example:

```
ADD         MACRO       SRC1,SRC2,DEST
```

defines a macro named **ADD** with three arguments.

Following the macro definition line, are the lines of text that define the macro. All lines following the macro are saved in a text buffer up to a line containing a **MACEND** or **ENDM** directive. This line terminates the macro definition.

NOTE: the **MACEND** or **ENDM** directive line may not contain a label.

For example:

```
ADD   MACRO       SRC1,SRC2,DEST
      CLC
      LDA         SRC1
      ADC         SRC2
      STA         DEST
      MACEND
```

defines a macro which adds the first two arguments and stores the result in the third.

Calling a Macro

To call a macro, all that is needed is to use the name of the macro just like an assembler directive followed by the macro arguments, if any. For example, the previously defined **ADD** macro could be invoked as:

```
ADD VAR1,#2,VAR1
```

which would add 2 to the contents of var1.

Redefining Assembler Directives and Opcodes

Since macro names are added to the assembler opcode and directive table, they will override an existing assembler directive or opcode. The assembler will generate an error if such a condition exists unless the **MACFIRST ON** option has been specified. Care should be taken when choosing macros names to avoid conflicts.

Macro Labels

Within a macro, there are several additional forms that labels may take. First, it is possible to concatenate two or more symbols, macro arguments and expressions using the `@' character. When the `@' is encountered, the current symbol ends and a new symbol begins, with the concatenation character discarded. The new symbol may be another symbol, a macro argument or the value of an expression. If an expression is specified, the expression must be enclosed in a balanced set of `<' and `>' characters. For example:

```
TST             MACRO       ARG

A@ARG           DB          0
B@<ARG>         DB          0
C@<2*ARG>       DB          0

        ENDM
```

defines a macro with one argument.

Within the macro body, a label is generated using the letter `A' followed by the literal value of the argument passed when the macro is invoked. The second label generated uses the letter `B' followed by the value of the argument. The third label generated uses the letter `C' followed by the value of the argument multiplied by 2.

Thus,

```
VAR     EQU     5
        TST     VAR
```

will generate:

```
TST     VAR
+               AVAR        DB          0
+               B5          DB          0
+               C10         DB          0
```

A second form of label only valid within a macro uses the `#' character to generate unique labels. If a macro defines a label within the macro, then calling the macro more than once will generate a duplicate label. This can be avoided by appending the `#' character to the end of the label name. The assembler will substitute a four digit number for the `#' character. This number is unique to each macro call.

For example, the program:

```
TEST    MACRO
LAB#    NOP
        JMP             LAB#
        ENDM

        TEST
        TEST
```

will generate the listing:

```
          1                    TEST      MACRO
          2                    LAB#      NOP
          3                              JMP     LAB#
          4                              ENDM
          5
          6                              TEST
+         6 00:0000: EA        LAB0001       NOP
+         6 00:0001: 4C xx xx                JMP     LAB0001
          7                              TEST
+         7 00:0004: EA        LAB0002       NOP
+         7 00:0005: 4C xx xx                JMP     LAB0002
```

Conditional assembly

Using conditional assembly, it is possible to have the same assembly language source produce different output depending on how certain elements are defined. A typical conditional consists of the conditional test followed by the statements that are assembled if the condition was true. These statements may be followed by an **ELSE** directive. If the **ELSE** is found, then the following statements are assembled if the condition was false. The conditional is terminated by either an **ENDIF** or **ENDC** directive.

For example:

```
IF              CNT>500
MESSG           CNT TOO HIGH!
ENDIF

IFTRUE  VAL>2
        LONG                    0
ELSE
        WORD                    0
ENDIF
```

Conditionals can be nested for more complex conditions. This example generates the proper size zero depending on the value of VAL.

```
IF              VAL=4
        LONG            0
ELSE
        IF              VAL=2
                WORD            0
        ELSE
                BYTE            0
        ENDIF
ENDIF
```

Each conditional test must be balanced by a corresponding **ENDC** or **ENDIF**. Note that all statements containing conditional tests or the directives **ELSE**, **ENDC**, and **ENDIF** may not have a label.

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 6 WDCxxAS (ASSEMBLER)

The WDC macro assembler, WDCxAS, provides all the tools and facilities to do professional assembly language program development.

Running the Program

The format for the assembler command is:

WDCxAS [-GSL1] [-I PATH] [-Dsym[=val]] [-O OUTPUT] SRCFILE

where *SRCFILE* is the name of the assembly language file that is to be translated. For example the command:

WDCxAS -O MYPROG.OUT MYPROG.ASM

will read the input file *MYPROG.ASM* and place the object code into the file *MYPROG.OUT*. If the **-O** option had not been specified, the result would have been placed in the file *MYPROG.OBJ* since *.OBJ* is the default output extension. If the *.ASM* extension had not been specified, the assembler would have looked for a file with no extension and then, if not found, would add the *.ASM* extension and tried again. Thus the simple command:

WDCxAS MYPROG

would assemble *MYPROG.ASM* and place the output in the file *MYPROG.OBJ*.

WARNING!: The line "WDCxxAS" must be less than 256 characters.

Option Summary

-1 Use version 1 control characters.
-D Define global equate.
-G Generate assembly source information.
-I Specify include directories.
-K Path name specifying name of listing file placed in __FILE__
-L Generate a listing file.
-O Specify the name of the output file.
-S Pass local symbols to linker.
-V Display the amount of RAM needed to assemble the program
-W Causes a change in the default page width to 132

Option Descriptions

-1

Version 1.0 of the assembler used slightly different control characters. In particular, the macro concatenation character `|' was changed to `@', the bit-wise inclusive OR character `^' was changed to `|', and the bit-wise exclusive OR character `^' was added. This option changes the characters back to the 1.0 versions for compatibility.

-D

This option is used to define an absolute symbol at run time. The symbol can then be used in conditional statements to change the code generated. The symbol can be followed with an equal sign and an absolute decimal value. If no value is specified, then the symbol is given the value one. For example the following command line:

WDCxAS -DCOUNT=3 -DDEBUG FILE.ASM

will set the absolute symbol **COUNT** to the value 3 and the symbol **DEBUG** to the value 1. Specifying these options is equivalent to the following statements appearing at the beginning of the source file:

```
COUNT   GEQU    3
DEBUG   GEQU    1
```

-G

The assembler and linker can now produce source level debugging information for programs. When this option is specified, the assembler will generate special object file records which indicate the number of bytes generated for each source line in the file. Information is also generated to determine the source file containing the source line. This option will increase the size of object modules generated by the assembler. The **INCDEBUG** directive can be used to control the generation of source information for included files. The –G option creates the .bin file used with the WDC Debugger (WDCDB.EXE) and the WDCDB.INI files.

-I

When the assembler encounters an **INCLUDE** or **APPEND** directive, the assembler looks in specific directories in a specific order for the named file. First, the current directory is checked. Next, any directories that have been specified using the **-I** option will be searched. Finally, if an environment variable called **WDC_INC_65816** or **WDC_INC_6502** has been defined, then any directories specified in that variable will be searched.

For example, if one of the following lines is in the *AUTOEXEC.BAT* file:

```
SET WDC_INC_65816=C:\WDC\INCLUDE;C:\WDC\MACROS
SET WDC_INC_6502=C:\WDC\INCLUDE;C:\WDC\MACROS
```

then, the command:

```
WDCxAS -I C:\MYINC PROG
```

will assemble the file *PROG.ASM*. If the file contains any **INCLUDE** or **APPEND** directives, then the assembler will look for the specified file in this order:

```
current directory
C:\MYINC
C:\WDC\INCLUDE
C:\WDC\MACROS
```

-K

This option causes the path name specifying the name of the listing file to be placed in the reserved word __FILE__.

-L

This option instructs the assembler to generate a listing file that will have the same root name as the output name and an extension of *.LST*. The format and output control of the listing file are controlled by assembler directives within the source file.

-O

This option is used to specify the name of the output file. Normally, the output file has the same root name as the source file, and the extension is changed to *.OBJ*. For example, the command:

```
WDCxAS MYPROG.ASM
```

will generate an output file called *MYPROG.OBJ*. If the **-O** option is used, the output file name can be specified directly. For example:

```
WDCxAS -O JUNK.REL MYPROG.ASM
```

will place the same output into a file called *JUNK.REL*.

-S

Normally, labels that aren't declared global are not placed in the object module since the assembler resolves all references to them. If the **-S** option is specified, the symbols are included in the object file so that the linker may pass them on to a symbol file that can be used when debugging.

-V

This is the Verbose option. This option displays the amount of RAM needed to assemble the program.

-W

This option causes a change in the default page width to 132, creating a wide listing.

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 7 WDCLN (LINKER)

The WDC linker, WDCLN, reads one or more object files and/or libraries and merges them into a single output file. References from one module to another are resolved during the link. The linker operates in two passes. In the first pass, each object module is scanned to determine what symbols are defined and what symbols are referenced. Symbols that are defined are entered into a symbol table. When other object modules want the address of a symbol they will look in the symbol table. If the symbol is not in the symbol table it is added to a list of undefined symbols. If a later module defines the symbol it is removed from the undefined list. If a library is encountered, it's dictionary is repeatedly scanned for any symbols that match any of the symbols in the undefined list. If such a symbol is found, the module that defines it is loaded from the library and its symbols are handled just like a normal object module. Through this process, only object modules that are needed are loaded from the library. Libraries are usually placed at the end of the list of object files. At the end of the first pass, all undefined symbol references should be resolved. During the second pass, the linker reads each object module a second time. As it reads each module, it generates the final output file based on the information in the object module.

Running the Program

Note: There are calls to user defined functions that are system dependent.
Example: _unlink, _ _close, _ _isatty,_ _write, _ _lseek, _ _fseek.
_ _read, _ _open, _ _creat, _ _ access (see WDC_SDS/INCLUDE/FCNTL.H

The WDCLN program is started by giving a command with the format:

```
WDCLN [-BEGNQTVWX] [-Hxx] [-Mxx] [-Sxx] [-O OUTPUT] [-Zsec=XX,XX]
      [-Asec=XX,XX] [-CXX,XX] [-DXX,XX] [-KXX,XX] [-UXX,XX]
      [-F argfile] OBJ1 [OBJ2 ...] [LIB1 ...] [-Lxx]
```

WARNING!: The line “WDCLN” must be less than 256 characters.

Input files are object files created by the WDCxAS assembler. The default extension for such an object file is *.OBJ*. If an object file is named without an extension, the linker first checks for the file without an extension and then adds the default extension. If the extension is specified, then the file is looked for only under that name. The linker looks for the input files in the current directory. If they are not found in the current directory, the linker looks at each of the directories defined in the **WDC_LIB** environment variable.

Placing a line such as:

```
SET WDC_LIB=C:\WDC\LIB
```

in the *AUTOEXEC.BAT* file will tell the linker where to look for common object files and libraries. Multiple directories may be separated by semi-colons. Libraries of object modules are created by the WDCLIB utility. Libraries can be specified using the full path and name of the file. Shorthand versions of libraries can also be specified using the **–L** option. The name of the output file is usually taken from the name of the first object module unless the **-O** option is specified. The extension on the object module name, usually *.OBJ*, is replaced by an extension appropriate to the hex format requested. For example, the command

```
WDCLN PROG.OBJ STUFF.OBJ CT.LIB
```

would create an output file called *PROG.S19* since Motorola S19 format is the default.

The output file can be explicitly specified by using the **–O** option. In that case, the command:

```
WDCLN -O TEST.HEX PROG STUFF -LCS
```

would place the program in the file called *TEST.HEX*. Note that even though the *.OBJ* extensions are omitted and the library is specified using the **-L** option, the arguments will reference the same files as in the previous example.

Option Summary

-A	Specify section address.
-B	Place bank info in .bnk file.
-C	Specify CODE address.
-D	Specify DATA address.
-E	Place errors in a .err file.
-F	Read arguments from the specified file.
-G	Generate source debug information.
-H	Specify the hex output format.
-J	Sort module info by name.
-K	Specify KDATA address.
-L	Specify a library name.
-M	Specify machine format.
-N	Discard .QCK symbols.
-O	Specify the name of the output file.
-P	Set the fill characters in the hex output file.
-Q	Tell the linker to be quiet.
-S	Specify the symbol file format to use.
-T	Generate a map file.
-U	Specify UDATA address.
-V	Display additional information.
-W	Disable warnings.
-X	Use EMM memory for symbol tables.
-Z	Set the spread for the section.

Option Descriptions

-A

This option is used to specify the relocation and ROM address of the named section. The option is followed by information in the following format:

```
-Asection=[XXXX][,[XXXX]]
```

where *section* is the name of the section to be located. The section name is followed by an `=' sign which in turn is followed by the relocation address and ROM address separated by a comma. All addresses are assumed to be hexadecimal numbers. If the comma and ROM address are not present, it is assumed that the relocation address will be used for the ROM address as well. If the comma is present and either the relocation address or the ROM address is missing, then the specified address is assumed to be the end of the previously specified section. For more information see CHAPTER 3.

EXAMPLE:

```
-Avec=FFE4          ;ROM and relocation both at FFE4
-Avec=FFE4,8000     ;in ROM at 8000, assembled as though at FFE4
-Avec=FFE4,         ;in ROM after previous section
-Avec=,8000         ;in ROM at 8000, assembled after previous
-Avec=,             ;ROM and relocation after previous
                    ;       this is the same as no option at all
```

-B

This option is used to create a file with the same root name as the output file and with the extension `**.BNK**'. This file is similar to the map file and contains bank information.

-C

This option is used to specify the relocation address and the ROM address of the predefined CODE section. The format is:

```
-C[XXXX][,[XXXX]]
```

which is similar to the **-A** option without the section name.

EXAMPLE:

```
-C8000              ;ROM and relocation both at 8000
-C18000,8000        ;in ROM at 8000, assembled as though at 18000
-C18000,            ;in ROM after previous section (0 for code)
-C,8000             ;in ROM at 8000, assembled at 0
-C,                 ;ROM and relocation both 0
                    ;this is the same as no option at all
```


-D

This option is used to specify the relocation address and the ROM address of the predefined DATA section. The format is:

-D[XXXX][,[XXXX]]

which is similar to the **-A** option without the section name.

EXAMPLE:

```
-D8000          ;ROM and relocation both at 8000
-D18000,8000    ;in ROM at 8000, assembled at 18000
-D18000,        ;in ROM after KDATA, assembled at 18000
-D,8000         ;in ROM at 8000, assembled after KDATA
-D,             ;ROM and relocation both after KDATA
                ;       this is the same as no option at all
```

-E

This option is used to create a file with the same root name as the output file and with the extension `**.ERR**'. This file contains any warnings or error messages generated during the link.

-F

This option causes the linker to continue reading options and file names from a file. When done, it then continues reading arguments from the command line. The name of the file follows the option **-F**. Lines beginning with a `#' character are ignored. For example, the following command links *PROG.OBJ* with *SUB1.OBJ*, ..., *SUB4.OBJ*, and *TC.LIB*. It reads some arguments from the file *PROG.LNK*:

WDCLN PROG.OBJ -F PROG.LNK TC.LIB

where *PROG.LNK* contains:

```
-O PROG.OUT
SUB1.OBJ SUB2.OBJ
SUB3.OBJ
SUB4.OBJ
```

WARNING: There is a limit of 5000 files for the source level information contained in reading file names from a file (Include files are counted in this total).

-G

This option tell the linker to generate source level information. When specified by itself with no additional symbol style option, the WDC symbol file format is generated. Otherwise, if the **-SN** Extended MicroTek symbol format option is specified, the source information is added as special records.

-H

This option is used to select the format of the hex output file. Four formats are currently supported. The following table shows the name of the format, the option used to generate it, the default file extension generated for the output file and the address field size.

Option	File	Size	Format
-HB	.BIN	--	Straight Binary
-HI	.HEX	16	Intel Hex
-HIE	.HEX	32	Extended Intel Hex
-HM19	.S19	16	Motorola S19
-HM28	.S28	24	Motorola S28
-HM37	.S37	32	Motorola S37
-HN	.ISX	24	Nintendo Binary
-HT	.TEK	16	Tektronix Hex
-HZ	.BIN	24	WDC Binary

Binary format:

The following binary format is generated if `-hz' is specified to the linker:

Initial byte 'Z' as signature.
Then for each block:
3 byte address
3 byte length
length bytes of data

The final block has an address and length of 0.
The default is Motorola S19.

-J

This option causes the module info to be placed in alphabetical order.
By default, module info is sorted by section.

-K

This option is used to specify the relocation address and the ROM address of the predefined KDATA section. The format is:

-K[XXXX][,[XXXX]]

which is similar to the **-A** option without the section name.

EXAMPLE:

```
-K8000              ;ROM and relocation both at 8000

-K18000,8000        ;in ROM at 8000, assembled at 18000
-K18000,            ;in ROM after CODE, assembled at 18000
-K,8000             ;in ROM at 8000, assembled after CODE
-K,                 ;ROM and relocation both after CODE
                    ;this is the same as no option at all
```

-L

This option takes the following characters and adds *.LIB* to form the name of the library. The default library directories specified in the **WDC_LIB** environment variable are then searched for the fully defined file name.

For example, the command:

WDCLN -J –LCL

will look for the file *LCL.LIB*.

Note: The order of the libraries is important! The linker will pull in the functions it needs from the ***first*** library it sees. For example, the following command:

WDCLN Sample.obj –LMS –LCS

will pull in the scanf and printf functions from the floating point library as it is specified first. This will result in larger code size! Therefore,

If you are using floating point math, put –LMS before –LCS so the proper functions are included.
If you are NOT using floating point math, do not include –LMS on the command line, or put it after –LCS.

Note: For the W65C02, use c.lib and/or m.lib. For the W65C816, use coc.lib, col.lib, com.lib, ms.lib, mm.lib, mc.lib, ml.lib, cs.lib, cm.lib, cc.lib, and/or cl.lib.

-M

This option is used to select a special machine mode. Currently, the only available machine modes are **-MN**, **-MN80** and **-MN21** which stand for Nintendo, slow and fast, and Nintendo Mode 21 respectively.

-N

If this option is specified, the linker will not place any symbols defined in a *.QCK* file into the symbol file. This is useful if the *.QCK* file is created from a large amount of data whose symbols are not required after linking. The symbol file can be significantly smaller if the data symbols are discarded.

-O

Option **-O** can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows **-O**. For example, the following command writes the executable program to the file *PROG.OUT*:

WDCLN -O PROG.OUT PROG.OBJ TC.LIB

If this option is not used, the linker derives the name of the executable file from that of the first input file with the extension changed to reflect the type of hex file being generated.

-P

This option sets the fill characters in the hex output file. The default, (no -P), does not add any fill characters to the hex output file. If this option is specified as –PFF, it will fill in the blank areas of the hex output file with $FF's (all 1's). If this option is specified as –P00, it will fill in the blank areas of the hex output file with zeros, (0's).

-Q

As the linker reads files and modules, it displays the name of each module. Each subsequent module name overwrites the preceding name. This option tells the linker not to display module names.

-S

This option controls the generation of symbol file information. By default, no symbol file is generated. When this option is specified, a symbol file is generated which can be used to aid in debugging the application.

The following table shows the options and the formats generated.

Option	Format
-S2	2500AD symbol format
-SM	MicroTek symbol format
-SN	Extended MicroTek symbol format
-SQ	Quick link object file
-SZ	WDC symbol format

The linker supports an extension to the Extended MicroTek symbol file format.

The linker generates the following additional symbol records
if the '-g' (source level info) option and '-sn' options have been selected.

0-9	Standard MicroTek symbol type - global symbols
50-59	Standard MicroTek symbol type - local symbols
101	Single character name that is the status Register as specified by LONGA/LONGI directives. In other words, if LONGA ON is specified a 101 record will be generated with a $20 as the ps value.
102	A two character name (low,high) that is the line number associated with this address.
103	The name is the source file name associated with the object module.
120+N	A zero length name with the address being the starting address for section N. Section 1 is CODE, section 2 is DATA, section 3 is UDATA. Other sections can probably be ignored unless you want to handle them.
150+N	A zero length name with the address being the ending address fot section N.

The accompanying program source, `nsym.c', will display the records of this symbol file format.

-T

This option instructs the linker to generate a text map file with the extension *.MAP*. The final address of each symbol is listed.

-U

This option is used to specify the relocation address and the ROM address of the predefined UDATA section. The format is:

-U[XXXX][,[XXXX]]

which is similar to the **-A** option without the section name. The UDATA section is a little different since it never needs to be in the ROM at all since it contains uninitialized data.

EXAMPLE:

```
-U8000          ;ROM and relocation both at 8000
-U18000,8000    ;in ROM at 8000, assembled at 18000
-U18000,        ;in ROM after DATA, assembled at 18000
-U,8000         ;in ROM at 8000, assembled after DATA
-U,             ;ROM and relocation both after DATA
                ;       this is the same as no option at all
```

-V

This option displays additional information to the screen giving the names of variables and their locations.
Note: If the linker output is longer than the screen will allow, the linker output can be redirected to a file and viewed in its entirety. You can redirect the linker output to a file by adding >> *filename.txt* to the end of the linker line.

For example:
WDCLN -c1000 –sz –hz –g –t –v –o exmpl1.bin t0s.obj exmpl1.obj –lcs >> output.txt
will redirect the linker output to the file output.txt. The file can then be viewed in a text editor.

-W **Note:** This option not used after V3.10

This option disables warnings from the linker. The linker will warn if a symbol defined in a program module overrides a symbol defined in a library module. This warning is useful for preventing hard to track down errors such as when the user defines a routine called *write* that overrides the library *write* routine.

-X

This option directs the linker to use EMM memory mapping to provide additional space for linking programs with large numbers of symbols. It must be the first option specified.

-Z

This option is used to specify the top and bottom address to use when spreading the indicated section of various modules across multiple banks of memory. The format is:

-Zsec=[bottom][,top]

The section specified by *sec* will be marked for spreading. The default bottom is 0 and the default top is \$1:0000. If no bottom or top is specified, the default is used. The first byte of the section is specified using the ROM and relative org directives. The following examples would spread code across the top and bottom 32K of each bank.

EXAMPLE:

-Zcode=8000 ; spread code starting at $8000
-Zcode=,8000 ; spread code from 0 to $8000

The sub-options for –Z are: -Zsec (section name)=, -Z code=, and –Zdata=,.

Quick Linking

Many programs are composed of a large amount of code and data. During development, large portions of the data will change only rarely. However, even a minute change will require the entire program to be relinked and then downloaded to the test platform. To alleviate this situation, the linker has the ability to link the data separately and generate a special object module which contains only the public symbols defined in the data link. This special object module is then linked with the remaining object modules each time a change is made. The binary file generated by the first link can be downloaded once and need not be downloaded again unless the data changes or becomes corrupted. To generate the special object module, the **-SQ** option is used. Instead of generating a symbol file, a file with a *.QCK* extension is created. This file is in object module format and contains all the symbols as ABSOLUTE equates. In addition, the **-N** option can be used when linking a *.QCK* file to prevent the symbols defined in the *.QCK* file from being placed in the symbol file. This can make the symbol file significantly smaller. For example, the following commands will create two WDC binary files the first of which will contain data and the second of which will contain the program code.

EXAMPLE:

WDCLN -SQ -HZ -O DATA.BIN DATA1.OBJ DATA2.OBJ DATA3.OBJ
WDCLN -HZ -N -O CODE.BIN DATA.QCK CODE1.OBJ CODE2.OBJ CODE3.OBJ

Technical Notes

Considerations for when CODE section exceeds bank 00

The linker will generate an error when the size of the CODE section exceeds bank 00 (or bank 01), even if the medium memory model is used.

To avoid this error, specify that the CODE ROM and relocatable address are the same as the DATA or KDATA ROM address:

-zCODE –C2000 –K2000

Since the CODE section is being spread, it will skip over any fixed locations such as DATA and KDATA.

Notes on the starting address for each section in the linker output

In some cases, the linker output will always display the same starting address for each section even if the first instruction is at the right address. For example:

Linker options:

```
-C2000
-ztst_conditional_test_on_char
-ztst_conditional_test_on_short
-ztst_function_branch
…….
…….
-Atst_conditional_test_on_char=2000
-Atst_conditional_test_on_short=2000
-Atst_function_branch=2000
…….
…….
-zKDATA
-K2000
-D200,
-U,
-bs
-hie
-v
-g
-sz
-t
-J
```

Linker output:

```
Section tst_function_branch:
    00002000 _BEG_TST_FUNCTION_BRANCH
    00002620 __module_1                    <= section start from here!!!
    00002652 __module_2
```

Since each module of TST_FUNCTION_BRANCH section may be spread over several memory banks, it is not possible to specify the start address of the section.
e.g.: module_1 may be allocated at $2500 and module_2 after next section because it does not fit into the available space in bank0.

Notes on creating a new DATA section

The C_STARTUP code clears the UDATA section. However, it may be required to have another DATA section (e.g. SAVE_DATA) that is NOT initialized (zeroed) at startup.

For example:

#pragma section UDATA=save_data

unsigned char my_save_data_section;

#pragma section CODE=user_code_section

void my_user_code_fuction(void) {}

#pragma section CODE=CODE
#pragma section UDATA=UDATA

LINKER COMMAND FILE:

-D200,
-Asave_data=, <- this section must be declared before any code section
-C2000 <- Start code in ROM/FLASH
-Zuser_code_section <- Set SPREAD option to keep user_code_section in the same memory bank
-Auser_code_section=2000<- Set same start addr. of CODE section
-ZKDATA <- set SPREAD option for KDATA
-AKDATA=2000 <- set same start addr.

Linker output:

```
Sections:
org=00002000 siz=000004DC end=000024DC 'CODE'
org=00002000 siz=00000103 end=00003503 'KDATA'
org=00003503 siz=0000005E end=00003561 'DATA'
org=00000000 siz=00000192 end=00000000 'UDATA'
org=00000000 siz=00000003 end=00000000 'save_data'
org=00002000 siz=00000028 end=00002504 'user_code_section'
----
----
org=0000FF00 siz=00000044 end=0000FF44 'startup'
org=0000FFA0 siz=00000024 end=0000FFC4 'hw_options'
org=0000FFC4 siz=00000020 end=0000FFE4 'ir_vectors'
org=0000FFE4 siz=0000001C end=00010000 'cpu_vectors´
----
----
Section: ORG:    ROM ORG:  SIZE:
CODE     002000  002000      4DCH (  1244)
KDATA    002000  002000      103H (   259)
DATA     000200  003503       5EH (    94)
UDATA    00025E  ------      192H (   402)
save_dat 0003F0  ------        3H (     3)
user_cod 002000  002000       28H (    40)
----
----
startup  00FF00  00FF00       44H (    68)
hw_optio 00FFA0  00FFA0       24H (    36)
ir_vecto 00FFC4  00FFC4       20H (    32)
cpu_vect 00FFE4  00FFE4       1CH (    28)
```

The new data section must be declared first. The **SPREAD** option must be used when new sections are declared to keep all of them within one memory bank, if they fit.

CHAPTER 8 WDCLIB (LIBRARIAN)

WDCLIB is a utility program that manipulates libraries of object modules. WDCLIB makes it possible to create a library of commonly used functions. This library can be very efficiently searched and any modules required by the program can be extracted from the library and placed in the output file.

Note: The standard libraries are in C:\WDC_SDS\Lib

Running the Program

The WDCLIB utility is started by giving a command with the format:

WDCLIB [-F ARGFILE] [-ADLSX] LIBRARY [OBJFILE ...]

where *LIBRARY* is the full or partial pathname of the library file to be created, read or modified. Since several object modules may be contained in the same original source file, WDCLIB keeps track of the name of the file that each module comes from. This allows all the modules associated with a file to be manipulated without tediously typing in the name of each module. Options may be specified individually or together.

Option Summary

-A add files to library
-D delete files from library
-F specify file with arguments
-L list files in library
-S list dictionary symbols
-X extract files from library

Option Descriptions

-A

This option tells WDCLIB to add the specified files to the library. The symbol dictionary is updated to include the names of symbols defined in the object modules in the files. If none of the options **-A**, **-D**, or **-X** are given, the default is to assume option **-A**. To create a library from a set of object files, use the command:

WDCLIB -A MYLIB.LIB LIBSRC1.OBJ LIBSRC2.OBJ LIBSRC3.OBJ

which will create a library file called *MYLIB.LIB* and add all the modules from the three object files. If *MYLIB.LIB* already existed, the modules from the three object files will be added to the library.

-D

The modules in the library that originally came from the named files are deleted from the library. Modules must be deleted before being replaced with new ones.

The following example shows how to remove the modules associated with an object file.

```
WDCLIB -D MYLIB.LIB LIBSRC3.OBJ
```

All of the modules associated with the file *LIBSRC3.OBJ* will be deleted from the library.

This example shows how to replace a file in a library.

```
WDCLIB -D MYLIB.LIB LIBSRC2.OBJ
WDCLIB -A MYLIB.LIB LIBSRC2.OBJ
```

The modules associated with *LIBSRC2.OBJ* will first be deleted from the library and then added from the new version of the file. The following options display information about the library file after the modification arguments, if any, have been processed.

-F

This option must be followed by the name of a text file. The file will be read and arguments will be extracted from the file. When the end of file is reached, additional arguments are again extracted from the command line. This allows more object modules than will fit on the standard command line to be processed at one time.

For example, these commands add all files with a `.OBJ' extension to the library.

```
DIR *.OBJ > OBJLIST
WDCLIB -A MYLIB.LIB -F OBJLIST
```

-L

This option causes a list of the files in the library to be printed. Associated with each file name is a file number. This number will also appear in the symbol listing which indicates which file contains the module that defines that symbol. This command will display the names of all files added to a library.

```
WDCLIB -L MYLIB.LIB
```

This command adds two files to the library.

```
WDCLIB -AL MYLIB.LIB LIBSRC1.OBJ LIBSRC2.OBJ
```

After the files are added, a list of all the files in the library will be printed.

-S

This option causes the dictionary of symbols contained in the library to be printed. The dictionary is printed in alphabetical order. The number of the file that defined the symbol along with the offset into the library of the module that defined it are printed beside the symbol name.

-X

The modules in the library that originally came from the named files are extracted from the library and placed into files with the same name. After extraction, the modules in the library are deleted.

The following example extracts two files from a library.

```
WDCLIB -X MYLIB.LIB LIBSRC3.OBJ LIBSRC1.OBJ
```

CHAPTER 9 WDCOBJ (EXAMINE OBJECT MODULES)

The WDCOBJ utility provides a means to examine object modules created by the WDCxAS assembler. WDCOBJ will print out the size and type of each section defined in the module, the names of all symbols defined or referenced by the object module, and if desired, each of the data records in the file. The WDCOBJ utility is of limited usefulness to the typical programmer and is included for completeness. Options allow control of the information displayed.

Running the Program

The WDCOBJ utility is started by giving a command with the format:

WDCOBJ [-DLRS] PATHNAME[.OBJ]

where *PATHNAME* is the full or partial pathname of the file that is to be examined. The file may be an object module produced by the WDC assembler or a library file. Options may be specified individually or together.

Option Summary

-D display debug info records
-L suppress data object records
-R display object records
-S suppress symbol information

Option Descriptions

-D

This option causes display of any source debug information records present in the object module. The appropriate options must have been specified when compiling or assembling for debug information to be present. The default is to NOT display debug information.

-L

Normally, when the records are displayed, all the data in the record is displayed in hexadecimal format. When option **-L** is specified, the data in the record is not displayed. This option is useful for examining the structure of a file without displaying all the individual data.

To examine the individual records in an object module but without seeing all of the data bytes, use the command:

WDCOBJ -L PROG.OBJ or WDCOBJ –L Prog

-R

This option causes display of each of the individual records in the object module. Information about the object file format is available on request.

To examine the individual records in an object module, use the command:

WDCOBJ -R PROG.OBJ or WDCOBJ –R Prog

-S

Normally, when WDCOBJ is run, the information for the sections is followed by the symbol information. When the **-S** option is specified, the symbol information is suppressed.
The following command displays just the section names and types of all modules in the file *PROG.OBJ*:

WDCOBJ -S PROG.OBJ

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 10 WDCSYM (EXAMINE SYMBOL TABLES)

The WDCSYM utility provides a means to examine symbol files generated by the WDCLN linker. WDCSYM will print out the sections defined in the target program and if desired the line tables, symbol records, auxiliary records and global symbols. Note: This is only for ZARDOZ symbol files. See the WDCLN manual and the –g and –sz options.

Running the Program

The WDCSYM utility is started by giving a command with the format:

WDCSYM [-ALS] PATHNAME[.SYM]

where *PATHNAME* is the full or partial pathname of the file that is to be examined. The file must be a symbol file produced by the WDC linker using the **-HZ** and **-G** options. Options may be specified individually or together.

Creating a batch file with the following line: WDCSYM filename.sym>>sym.txt, will create the file sym.txt that can be read in Notepad.

Option Summary

-A display auxiliary table
-L display line tables
-S display global symbols

Option Descriptions

-A

This option causes the display of the auxiliary table. This table contains typing information, array sizes and other information used in source level debugging. Only one table is present in the symbol file and is referenced by all sections and modules.

-L

This option causes the display of the line information data for each section in the symbol file.

-S

This option causes the display of all global symbol records in the symbol file. Normally, the global symbol records are suppressed.
The following command displays the section information, and symbols for all sections and the global symbols as well.

WDCSYM -S PROG.SYM

The basic structure of the file is outlined as follows:

File Header
Module 1 Information
    Section 1 Information
    …
    Section N Information
    Line Record Information
    Symbol Record Information
Module 2 Information

…
Module N Information
…
Global Symbol Records
String Table
Auxiliary Record Table
Source File Information
End of file

Note: See Chapter 6 of the Simulator/Debugger manual for more information.

CHAPTER 11 Assembly Opcodes

Standard Instructions

The following is a list of all the standard W65C816 opcode mnemonics.

adc	and	asl	bcc
bcs	beq	bit	bmi
bne	bpl	bra	brk
brl	bvc	bvs	clc
cld	cli	clv	cmp
cop	cpx	cpy	dec
dex	dey	eor	inc
inx	iny	jml	jmp
jsl	jsr	lda	ldx
ldy	lsr	mvn	mvp
nop	ora	pea	pei
per	pha	phb	phd
phk	php	phx	phy
pla	plb	pld	plp
plx	ply	rep	rol
ror	rti	rtl	rts
sbc	sec	sed	sei
sep	sta	stp	stx
sty	stz	tax	tay
tcd	tcs	tdc	trb
tsb	tsc	tsx	txa
txs	txy	tya	tyx
wai	wdm	xba	xce

Alternate Instructions

The following is a table of less common aliases for standard instructions.

Alias	Standard
bge	bcs
blt	bcc
cpa	cmp A
dea	dec A
ina	inc A
ret	rts
swa	xba
tad	tcd
tas	tcs
tda	tdc
tsa	tsc
xor	eor

W65C02S Instructions

The following instructions are enabled when **CHIP W65C02** is enabled.

bbr0	bbr1	bbr2	bbr3
bbr4	bbr5	bbr6	bbr7
bbs0	bbs1	bbs2	bbs3
bbs4	bbs5	bbs6	bbs7
rmb0	rmb1	rmb2	rmb3
rmb4	rmb5	rmb6	rmb7
smb0	smb1	smb2	smb3
smb4	smb5	smb6	smb7

Addressing Modes

This section provides a brief description of the 24 allowable addressing modes.

IMMEDIATE	OPCODE	VALUE

The operand is the second byte in 8-bit mode or the second and third byte in 16-bit mode.

ABSOLUTE	OPCODE	ADDR
	OPCODE	|ADDR

The operand is an address composed of the Data Bank register as the high-order 8 bits of a 24-bit address. The low-order 16 bits come from the second and third bytes of the instruction.

ABSOLUTE LONG	OPCODE	>ADDR

The operand is a 24-bit address that comes from the next three bytes of the instruction.

DIRECT	OPCODE	<ADDR

The operand is an address in Bank 0 that comes from adding the second byte of the instruction to the Direct Page register.

ACCUMULATOR	OPCODE	A

The operand is the accumulator. The effect is 8-bit if the status register M bit is 1 otherwise the effect is 16-bit. This instruction is always one byte.

IMPLIED	OPCODE

There is no operand for this addressing mode. The instruction is always one byte.

DIRECT INDIRECT INDEXED	OPCODE	(<ADDR),Y

The second byte of the instruction is added to the Direct Page register to determine the location in Bank 0 of a 16-bit address that is combined with the Data Bank register to form a 24-bit address. The Y index register is added to this address to form the final address. If the status register X bit is a 1, then 8 bits of the Y register are added otherwise 16 bits are added.

DIRECT INDIRECT LONG INDEXED OPCODE [<ADDR],Y

The second byte of the instruction is added to the Direct Page register to determine the location in Bank 0 of a 24-bit address. The Y index register is added to this address to form the final address. If the status register X bit is a 1, then 8 bits of the Y register are added otherwise 16 bits are added.

DIRECT INDEXED INDIRECT OPCODE (<ADDR,X)

The second byte of the instruction is added to the sum of the X register and the Direct Page register to form the Bank 0 address of a 16-bit address that is combined with the Data Bank register to form a 24-bit address.

DIRECT INDEXED WITH X OPCODE <ADDR,X

The second byte of the instruction is added to the sum of the X register and the Direct Page register to form an address in Bank 0.

DIRECT INDEXED WITH Y OPCODE <ADDR,Y

The second byte of the instruction is added to the sum of the Y register and the Direct Page register to form an address in Bank 0.

ABSOLUTE INDEXED WITH X OPCODE ADDR,X
OPCODE |ADDR,X

The second and third bytes of the instruction are combined with the Data Bank register to form a 24-bit address. The X register is added to form the final 24-bit address.

ABSOLUTE LONG INDEXED WITH X OPCODE >ADDR,X

The second, third and fourth bytes of the instruction form a 24-bit address which is added to the X register to form the final address.

ABSOLUTE INDEXED WITH Y OPCODE ADDR,Y
OPCODE |ADDR,Y

The second and third bytes of the instruction are combined with the Data Bank register to form a 24-bit address. The Y register is added to form the final 24-bit address.

PROGRAM COUNTER RELATIVE BRANCH LABEL

The second byte of the instruction is added to the value of the program counter after the program counter has been updated to point at the next instruction. The byte is considered a signed quantity. The resulting address is used as the new program counter. The Program Bank register is not affected.

PROGRAM COUNTER RELATIVE LONG	BRL	LABEL
	PER	LABEL

The second and third bytes of the instruction are added to the value of the program counter after the program counter has been updated to point at the next instruction. The word is considered a signed quantity. The resulting address is used as the new program counter for the **BRL** instruction. The Program Bank register is not affected. With the **PER** instruction, the resulting address is pushed onto the stack.

ABSOLUTE INDIRECT	OPCODE	(ADDR)
	OPCODE	(| ADDR)

The second and third bytes of the instruction form a 16-bit address in Bank 0. The 16-bits at the specified address are loaded into the Program Counter. If the opcode is **JML**, the third byte at the address in Bank 0 is loaded into the Program Bank register.

DIRECT INDIRECT	OPCODE	(<ADDR)

The second byte of the instruction is added to the Direct Page register to form an address in Bank 0. The 16-bit value at the Bank 0 address is combined with the Data Bank register to form a 24-bit address.

DIRECT INDIRECT LONG	OPCODE	[<ADDR]

The second byte of the instruction is added to the Direct Page register to form an address in Bank 0. The 24-bit value at the Bank 0 address is used as the final address.

ABSOLUTE INDEXED INDIRECT	OPCODE	(ADDR,X)
	OPCODE	(| ADDR,X)

The second and third bytes of the instruction form a 16-bit address that is added to the X register to form a 16-bit address in Bank 0. The 16-bit address at the specified Bank 0 location is loaded into the Program Counter. The Program Bank register is not changed.

STACK	OPCODE	

Stack addressing refers to all instructions that push or pull data on or off the stack. It is a special case of IMPLIED.

STACK RELATIVE	OPCODE	<ADDR,S

The second byte of the instruction is added to the stack pointer to form a 16-bit address in Bank 0.

STACK RELATIVE INDIRECT INDEXED	OPCODE	(<ADDR,S),Y

The second byte of the instruction is added to the stack pointer to form a 16-bit address in Bank 0. The 16-bit value at the address is combined with the Data Bank register to form a 24-bit address that is added to the Y register to form the final 24-bit address.

BLOCK	MVN	DST,SRC
	MVP	DST,SRC

The second byte of the instruction is used as the destination bank number with the Y index register being the low-order 16 bits. The third byte is the source bank number with the X index register providing the low-order 16 bits.

Special Cases

A few of the 65816 opcodes will accept more than one form of address. Each of the instructions and the alternates is listed below.

MVN	#src,#dst
MVN	src,dst
MVP	#src,#dst
MVP	src,dst
PEA	absolute
PEA	#value
PEI	(direct)
PEI	direct
PEI	#direct
PER	label
PER	#offset
JSR	> address
JSL	> address
JML	> address
JMP	> address
JML	(absolute)
JMP	[absolute]

THIS PAGE LEFT INTENTIONALLY BLANK

CHAPTER 12 Assembly Directives

File and Symbol Control

These directives control the organization of files, modules, sections and symbols. See the overview for a more detailed discussion of modules and sections.

WARNING: When using long paths with spaces embedded in the path name, enclose the name in quotes! For example, APPEND "F\WDC\SAMS PROJECTS\MACROS.INC"

APPEND [LABEL] APPEND FILENAME

This directive causes the assembler to stop reading the current file and to read lines from the specified file instead. The original file is not returned to. Thus, this is usually the last statement in a file since any following it are ignored. The line number counter is not reset. Filenames are read up to a space, tab, semi-colon or end of line. Filenames may also be enclosed in single or double quotes.

EXAMPLE:

```
APPEND      c:\src\asmend.asm      ;use the common ending
```

INCLUDE [LABEL] INCLUDE FILENAME

This directive reads assembly language statements from the specified file. When the end of the file is reached, or an **END** directive is parsed, the assembler continues with the line following the **INCLUDE** directive. The line numbers are started again at 1 with each file that is included. Filenames are read up to a space, tab, semi-colon or end of line. Filenames may also be enclosed in single or double quotes.

EXAMPLE:

```
INCLUDE     c:\src\macros.inc      ;load macros
```

INSERT [LABEL] INSERT FILENAME

This directive reads a binary image from the specified file and inserts it into the object module at the current program counter. The assembler continues with the line following the **INSERT** directive. Filenames are read up to a space, tab, semi-colon or end of line. Filenames may also be enclosed in single or double quotes. When searching for files, the path specified by the **-I** option is used.

EXAMPLE:

```
INSERT      c:\src\sounds.inc      ; insert sound data
```

END [LABEL] END [VALUE]

When the **END** directive is encountered in a source file, no further lines are read. When the **END** directive is encountered in an included file, the file is closed and the next line is read from the previous include or source file. If *VALUE* is specified it is used as the starting address of the program if the output format has a record type that supports it.

EXAMPLE:

```
MAIN:   RTS     ; a very short program
        END     ; the end of the program

I can put anything here because the assembler never reads it.
```

EXIT [LABEL] EXIT TEXT

This directive displays the message to the output terminal and then causes the assembler to exit. This is typically used in a conditional when some event triggers it.

EXAMPLE:

```
NUMSYMS SET     NUMSYMS+1           ;add one more symbol
        IF      NUMSYMS>500         ;too many symbols?
        EXIT    Symbol table overflow!
        ENDIF
```

MODULE MODULE MODNAME
ENDMOD ENDMOD

These directives are used to mark the beginning and end of an independent module of a program. Each module acts like an independent file with a new symbol table. Modules are used almost exclusively when creating libraries of functions. They allow multiple functions to be defined in the same source file. See the overview for a more detailed description of how modules are used.

EXAMPLE:

```
MODULE      COPYSTR     ;function to copy a string
...                     ;function body
ENDMOD                  ;end of module
```

SECTION LABEL SECTION OPTIONS

This directive is used to define a new section type. There are five pre-defined sections with the names **PAGE0**, **CODE, KDATA**, **DATA**, and **UDATA**. **CODE** is the default. See CHAPTER 3 for a discussion of these sections. Up to 250 sections may be defined. The name of the section is taken from the *LABEL*. Additional options may be specified to determine the type of section being defined.
OFFSET ADDR This option makes the section an ABSOLUTE section which starts at the specified *ADDR*. It is equivalent to defining a section followed by an **ORG** directive.

INDIRECT ADDR This is similar to **OFFSET**, but tells the linker to assemble the code at the specified *ADDR*, but to actually place it somewhere else. This is used when generating data into ROM that will be copied to another address.

REF_ONLY This option indicates that any labels in the section are to be recorded at the correct offset, but no actual data is generated. This is useful for creating templates or when uninitialized data is being created. Both the **PAGE0** and **UDATA** sections default to being **REF_ONLY**. Once a section has been defined, its name is added to the opcode table. Thus, the same name can be used as a symbol as well. To add code or data to a section, the name of the section can be used as a directive. The **SECTION** definition automatically causes a switch to the new section.

EXAMPLE:

```
MYDATA    SECTION    OFFSET $10000    ;define section in bank 1
MYDATA:   DB         $1               ;save signature byte
          ENDS                        ;back to previous section

          LDA        >MYDATA          ;get signature byte
          CMP        #1

          MYDATA                      ;switch to MYDATA again
          RMB        20               ;save some space
          DB         2                ;set number of widgets
          ENDS                        ;back to previous section
```

ENDS ENDS

This directive is used to undo the effect of the previous section changing directive. Sections are nested, with each section name directive nesting a bit deeper. Sections can be nested up to 500 deep. Each **ENDS** directive ``un-nests" one level.

EXAMPLE:

```
NOP          ;start out in CODE section
   DATA      ;switch to DATA
   DB   1
     UDATA   ;switch to UDATA
     RMB  3
     ENDS    ;back to DATA
   DB   2
   ENDS      ;back to CODE
RTS
```


ORG	[LABEL]	ORG	OFFSET
ORIGIN	[LABEL]	ORIGIN	OFFSET

This directive does two things. First, it marks the current section as absolute. Second, it sets the program counter to the specified value. All subsequent labels are defined with an absolute address. When switching sections, the **ORG** statement should follow the statement that switches sections.

EXAMPLE:

```
        DATA                ;switch to the data section
        ORG     $200        ;set pc to hex 200
        ...
        CODE
        ORIGIN  $12000      ;set code pc to 01:2000H
```

EQU	LABEL	EQU	EXPRESSION
EQUAL	LABEL	EQU	EXPRESSION

These directives set the label to the value specified by the expression. The expression may contain only one level of indirect reference. The label has type ABSOLUTE unless equated with an expression containing a relocatable label.

EXAMPLE:

```
LAB1    EQU     100         ;absolute label
LAB2    EQUAL   MAIN+4      ;relocatable if MAIN is relocatable
```

GEQU	LABEL	GEQU	ABSOLUTE

This directive is similar to the **EQU** directive with two differences. First, the expression *must* be an absolute numerical value. No forward reference is allowed. The second difference only affects programs which use the **MODULE** directive. Symbols defined by the normal **EQU** directive are cleared at the beginning of each module. Symbols defined using the **GEQU** directive are retained across modules.

EXAMPLE:

```
LAB1    GEQU   100          ;absolute label
LAB2    GEQU   LAB1+4       ;also absolute
```

DEFL	LABEL	DEFL	EXPRESSION

SET	LABEL	SET	EXPRESSION
VAR	LABEL	VAR	EXPRESSION

These directives are similar to the **EQU** directive in that they set a label to a particular value. However, with these directives, the value can be changed with a later directive. The argument must be an absolute expression.

EXAMPLE:

```
CNT     DEFL    1               ;initialize counter
CNT     SET     CNT+1           ;increment counter by 1
CNT     VAR     EDATA-BDATA     ;difference is always absolute
```

EXTERN	[LABEL]	EXTERN	LABEL[,LABEL...]
EXTERNAL	[LABEL]	EXTERNAL	LABEL[,LABEL...]
XREF	[LABEL]	XREF	LABEL[,LABEL...]

These directives declare the specified symbol(s) to be external to the current file. Any symbols which are not defined in the current file or module and that need to be referenced must be marked external otherwise an undefined symbol error will be given at the end of Pass 1. A single symbol name may be specified, or m
ultiple symbols may be specified separated by commas.

EXAMPLE:

```
        EXTERN      copystr         ;say copystr defined elsewhere
        LDA         #ADDR1          ;get source address
        LDX         #ADDR2          ;get destination address
        JSL         copystring      ;copy the string
        EXTERN      Page0 IO_UART   ;give IO_UART a Page0 attribute
```

EXTERNS	[LABEL]	EXTERNS	ON
	[LABEL]	EXTERNS	OFF

When this directive is turned on, all labels that are undefined are automatically made public without using the **EXTERN**, **EXTERNAL** or **XREF** directives. This directive can be used to force all undefined symbols to be external without having to do so on an individual basis. The linker will then attempt to find matches for the symbols in other object modules.
The default is **OFF**.

EXAMPLE:

```
        lda         LAB1        ;an undefined label
        EXTERNS     ON          ;turn on automatic extern
        lda         LAB2        ;undefined, but external
```

GLOBAL	[LABEL]	GLOBAL	LABEL[,LABEL...]
PUBLIC	[LABEL]	PUBLIC	LABEL[,LABEL...]
XDEF	[LABEL]	XDEF	LABEL[,LABEL...]

These directives are similar to the previous set except that these indicate symbols that are defined in the current file and are referenced by a different file. If a label is not declared global, then it will not be listed in the object module and can not be found by the linker.

EXAMPLE:

```
        PUBLIC    copystr     ;tell assembler other files can call
copystr STA       <1          ;save in direct page
        STX       <3          ;save second pointer
loop    LDA       (<1)        ;get byte
        STA       (<3)        ;copy it
        BEQ       done        ;loop if non-zero
        INC       <1          ;bump address
        INC       <3
        BRA       loop        ;continue till done
```

GLOBALS	[LABEL]	GLOBALS	ON
	[LABEL]	GLOBALS	OFF

When this directive is turned on, all labels that are defined are automatically made global without using the **GLOBAL**, **PUBLIC** or **XDEF** directives. This includes any symbols defined using **EQU** or **SET** directives. This may be a useful thing to do when debugging a program since the linker can generate a symbol table with all symbol addresses. Only global symbols are passed to the linker, so placing this directive as the first thing in the source file will make all labels global and cause them to appear in the symbol table.
The default is **OFF**.

EXAMPLE:

```
LAB1:
        GLOBALS   ON          ;not a global symbol
LAB2:                         ;turn on all globals
                              ;this symbol IS global
```

MESSAGE	[LABEL]	MESSAGE	TEXT
MESSG	[LABEL]	MESSG	TEXT

This directive displays the indicated message text during Pass 2 of the assembly process. One use would be inside of conditionals to indicate that a certain set of conditions exist. Another use is as a reminder of what options to use when linking.

EXAMPLE:

```
IF      NUM>10
MESSAGE  More than 10 widgets!
ENDIF
MESSG    Don't forget to use -HM28!
```


EFUNC EFUNC

This directive serves only as a placeholder to mark the end of a C function generated by the compiler. It is used by the post-pass peephole optimizer that works on a single function at a time.

EXAMPLE:

```
        EFUNC
```

INCDEBUG	[LABEL]	INCDEBUG	ON
	[LABEL]	INCDEBUG	OFF

When the assembler is directed to generate source level information, it does so for the original source file and any included source files. Since many included source files contain only equates and symbol definitions, source information isn't very useful and takes additional space and time to produce. When this directive is turned on, source information is generated for included files. If **INCDEBUG** is off, no source information is produced. This directive cannot be nested. The default is **ON**.

EXAMPLE:

```
INCLUDE  "j1.inc"      ;source info for this file
INCDEBUG OFF           ;turn off source info
INCLUDE  "j2.inc"      ;no source info for this file
```

Parsing Control

The directives in this section control how the opcodes and opcode arguments are parsed.

CASE	[LABEL]	CASE	ON
	[LABEL]	CASE	OFF

This directive controls whether symbol names are case sensitive or not. If the directive is **ON**, all symbol names are recorded exactly as defined. If the directive is **OFF**, all symbol names are mapped to lower case. The default is **ON**.

EXAMPLE:

```
Lab1:
        BRA     LAB1        ;generates an error

        CASE    OFF
Lab2:
        BRA     LAB2        ;no error generated
```

CHIP

[LABEL]	CHIP	65C02
[LABEL]	CHIP	W65C02S
[LABEL]	CHIP	65816

This directive controls which set of opcodes and addressing modes is to be used. The valid values for type are 65C02, W65C02S, and 65816. The W65C02S option enables the extra BBRx, BBSx, RMBx and SMBx instructions. The default for WDC816AS is **65816**. The default for WDC02AS is **W65C02S**. The Rockwell version of the W65C02S does not support the WAI and STP instructions.

EXAMPLE:

```
CHIP    65816       ;start in native mode
SEC                 ;set carry for emulation mode
XCE                 ;go into emulation mode
CHIP    65C02       ;don't want to use any 65816 codes
```

CHKIMMED	[LABEL]	CHKIMMED	ON
	[LABEL]	CHKIMMED	OFF

This directive controls whether an error is generated when an immediate load to a register is larger than will fit in the register. Legal values range from -127 to 255 for a short register and -32767 to 65535 for a long register. The default is **OFF**, which will not generate an error.

EXAMPLE:

```
LONGA       OFF
CHKIMMED    OFF

LDA         #$101       ; no error generated

CHKIMMED    ON
LDA         #$101       ; generates an error
```

COMMENT	[LABEL]	COMMENT	CHAR

This directive is used to specify a block of lines as all being comments. The character argument of the **COMMENT** directive is used as an end marker. Lines are treated as comments until a line is encountered which contains the end marker character.

EXAMPLE:

```
COMMENT #
these lines are
all just comments
this is the last line #
```


DBREG [LABEL] DBREG [VALUE]

This directive is used to indicate to the assembler the run-time value of the Data Bank register. Normally, when the **DBREG** directive has been used, if a symbol is referenced without a specific addressing mode, the assembler will generate long absolute reference. When this directive is used, the assembler can check for references to absolute symbols that are defined in the indicated data bank and generate a two-byte absolute reference instead.

EXAMPLE:

```
A       EQU     $1:0000
B       EQU     $2:0000
        DBREG   $2
        LDA     A       ;long absolute address used
        LDA     B       ;absolute address used
```

DPAGE [LABEL] DPAGE [VALUE]

This directive is used to indicate to the assembler the run-time value of the Direct Page register. Normally, when the **DPAGE** directive has been used, if a symbol is referenced without a specific addressing mode, the assembler will generate an absolute reference. When this directive is used, the assembler can check for references to absolute symbols that are defined in the current direct page and generate a one-byte direct page reference instead.

EXAMPLE:

```
A       EQU     $24
B       EQU     $34
        DPAGE   $30
        LDA     A       ;absolute address used
        LDA     B       ;direct page address used
```

LONGA [LABEL] LONGA ON (Default)
[LABEL] LONGA OFF

This directive is used to indicate to the assembler the size of the accumulator. This tells the assembler to generate 16 bit versus 8 bit immediate values when the accumulator is involved. The ONLY effect this directive has concerns immediate operands. The X and Y registers are not affected.
The default is **ON**.

EXAMPLE:

```
        LONGA   ON
        LDA     #0      ;this will generate two bytes of zero
        LONGA   OFF
        LDA     #0      ;this will generate one byte of zero
```

LONGI	[LABEL]	LONGI	ON	(Default)
	[LABEL]	LONGI	OFF	

This directive is used to indicate to the assembler the size of the X and Y index registers. This tells the assembler to generate 16 bit versus 8 bit immediate values when these registers are involved. The ONLY affect this directive has concerns immediate operands. The accumulator is not affected.
The default is **ON**.

EXAMPLE:

```
LONGI   ON
LDX     #0      ;this will generate two bytes of zero
LONGI   OFF
LDY     #0      ;this will generate one byte of zero
```

RADIX	[LABEL]	RADIX	NUM
	[LABEL]	RADIX	CHAR

This directive changes the default radix for numbers in the operand field of instructions and directives. The default radix is decimal, so numbers that do not have a binary, octal or hexadecimal qualifier will be interpreted as decimal numbers. For example, the number `10' can be either 2, 8, 10, or 16 depending on the default radix. The radix can be specified as the number of the base, or by the letter that acts as the qualifier for that base. The following table summarizes the possible choices for the **RADIX** directive.

B	2	Binary
O,Q	8	Octal
D	10	Decimal
H	16	Hexadecimal

Note that the arguments to the **RADIX** directive are always assumed to be *decimal* numbers. If the radix is set to hexadecimal, then it is not possible to indicate binary or decimal numbers by using a trailing **B** or **D** since they will be considered part of the hexadecimal number.

EXAMPLE:

```
RADIX   Q       ;octal base
LDA     #16     ;octal 16 == decimal 14
LDA     #16D    ;decimal 16 == decimal 16
RADIX   16      ;hexadecimal base
LDA     #16     ;hex 16 == decimal 22
LDA     #16D    ;hex 16D == decimal 365
```

SPACES	[LABEL]	SPACES	ON	(Default)
	[LABEL]	SPACES	OFF	

This directive determines whether spaces or tabs are allowed between the elements of an instruction operand. If **SPACES** are **OFF**, then there can be no blanks or tabs between parts of the operand. Otherwise, the first space or tab will be interpreted as the end of the operand and the rest of the line will be treated as a comment.

EXAMPLE:

```
        LDA  ADDR , X
```

will be interpreted as just:

```
        LDA  ADDR
```

with `, X' as a comment.

If **SPACES** are **ON**, then the preceding example would be handled correctly. However, consider the following example:

```
        LDA  ADDR /NOTE
```

With **SPACES** turned **OFF**, the `/NOTE' would be considered a comment. With **SPACES** turned **ON**, the assembler will assume that the value of `ADDR' should be divided by the value of `NOTE'. When **SPACES** are **ON**, you must always use a semi-colon to begin a comment.

The default is **OFF**.

LLCHAR [LABEL] LLCHAR CHAR

This directive changes the character that is used to denote a temporary label. The default is the `?' character.

EXAMPLE:

```
        LLCHAR      /
/1      NOP
        BRA         /1
```

Data Definition Control

These directives are used to place data into the output file. The first few directives are used to affect individual bits of string operands. Data can be placed as bytes, words, or long words.

BIT7 [LABEL] BIT7 ON
[LABEL] BIT7 OFF

When enabled, this directive causes bit 7 of any bytes generated by the **ASCII** directive or by string arguments to the **BYTE**, **DB**, **DEFB**, **FCB** and **STRING** directives. The default is **OFF**.

EXAMPLE:

```
        DB      'A'         ;generates a hex 41 byte
        BIT7    ON
        DB      'A'         ;generates a hex C1 byte
```

MASK [LABEL] MASK AND_VALUE,OR_VALUE,SUB_VALUE

This directive allows precise bit control over all characters generated by a string argument to a data directive. Each byte in a string argument is bitwise-ANDed with the *AND_VALUE* and then bitwise-ORed with the *OR_VALUE* and then the *SUB_VALUE* is subtracted. The default is *AND_VALUE* = **$FF**, *OR_VALUE* = **$00**, and *SUB_VALUE* = **$00**.

EXAMPLE:

```
        DB      'A'             ;generates a hex 41 byte
        MASK    $FF,$20         ;convert to lower case
        DB      'A'             ;generates a hex 61 byte (`a')
        MASK    $DF,$00         ;convert to upper case
        DB      'b'             ;generates a hex 42 byte (`B')
        MASK    $FF,$00,$41     ;convert to alpha offset
        DB      'G'             ;generates a hex 7
```

SQUOTE	[LABEL]	SQUOTE	ON
	[LABEL]	SQUOTE	OFF

This directive controls how quoted strings are handled when used as the argument to any of the data definition directives. When **SQUOTE** is **ON**, a single quote character begins the string and the string continues to the end of the line. When **SQUOTE** is **OFF**, a second single quote is required to terminate the string. The default is **OFF**.

EXAMPLE:

```
        DB      'this line is ok'
        DB      'this line will generate an error
        SQUOTE  ON
        DB      'this line is now ok
```

TWOCHAR	[LABEL]	TWOCHAR	ON
	[LABEL]	TWOCHAR	OFF

This directive enables certain two character combinations enclosed in quotes to be interpreted as a single non-printable character. The following table displays the combinations, their hex value and what they represent.

CR	$0D	Carriage return
HT	$09	Horizontal tab
LF	$0A	Line feed
NL	$00	Null
SP	$20	Space

The default is **OFF**.

EXAMPLE:

```
                                ; the following line starts with a tab
        TWOCHAR ON
LINE:   DB      'HT','A line','CR','LF','NL'
                                ; and ends with a standard line ending
                                ; and a terminating null
```


ASCII ASCII TEXT
ASC ASC TEXT

This directive stores it's argument in successive bytes of memory. The string starts with the first character that is not a blank or a tab and continues till either the end of the line or the | character is encountered. This means that comments will be taken as part of the string unless the | character is used. The string does not need to be enclosed within ' characters.

EXAMPLE:

```
ASCII  This is a short string.      ;Not part of string!
ASCII  This is a test.              ;This is part of the string too!
```

FCC [LABEL] FCC CHAR TEXT CHAR

This directive stores it's string argument in successive bytes of memory. This directive allows more control than the **ASCII** directive in beginning and terminating its argument. The first non-blank character found is read and used as the terminating character of the string. The terminating characters are not considered part of the string.

EXAMPLE:

```
FCC     /this is a 'string'/        ;easy way to include quotes
```

DATE [LABEL] DATE

This directive outputs the bytes that correspond to the current date in the format:

DDD MMM DD YYYY HH:MM

where DDD is the day of the week, MMM is the month, DD is the day of the month, YYYY is the year, and HH:MM is the time in hours and minutes.

EXAMPLE:

```
FCC     /Today's date is /      ;first part
DATE                            ;second part
DB      $D,$A,0                 ;last part
```

DA [LABEL] DA [VALUE,...]

This directive is used to generate a three-byte address. Multiple values may be used, separated by commas. If no values are specified, three null bytes are generated.

EXAMPLE:

```
DA                  ;generate three null bytes
DA      1           ;generate three bytes (1,0,0)
DA      LAB         ;generate three-byte address
DA      LAB1,LAB2   ;generate two three-byte addresses
```

BYTE	[LABEL]	BYTE	[VALUE,...]
DB	[LABEL]	DB	[VALUE,...]
DEFB	[LABEL]	DEFB	[VALUE,...]
FCB	[LABEL]	FCB	[VALUE,...]
STRING	[LABEL]	STRING	[VALUE,...]

These directives generate byte data. Multiple values may be used, separated by commas. A value may also be a string of characters enclosed in either single or double quotes. If no values are specified, one null byte is generated.

EXAMPLE:

```
        BYTE            ;generate a single null byte
        DB      1       ;generate a byte with value $01
        DEFB    1,'abc',0       ;generate 5 bytes
        FCB     LAB,0   ;generate 2 bytes
        STRING  1,2,3   ;works just like BYTE
```

DC	[LABEL]	DC	[VALUE,...]

The **DC** directive is very similar to the **BYTE** directives. However, the last byte generated by the **DC** directive will have the high order bit set. This is often used to indicate the end of a string. If no values are specified for the **DC** directive, one byte of $80 is generated.

EXAMPLE:

```
        DC      'abc'   ;generates $61,$62,$E3
```

DEFW	[LABEL]	DEFW	[VALUE,...]
DW	[LABEL]	DW	[VALUE,...]
FDB	[LABEL]	FDB	[VALUE,...]
WORD	[LABEL]	WORD	[VALUE,...]

These directives generate word data. Each argument generates two byes with the low byte first followed by the high byte. Multiple values may be used, separated by commas. If no argument is specified, then a null word is generated.

EXAMPLE:

```
        DEFW            ;generates $00,$00
        DW      1       ;generates $01,$00
        FDB     1,2     ;generates $01,$00,$02,$00
```

DBYTE	[LABEL]	DBYTE	[VALUE,...]

This directive also generates word data. However, this directive stores the high byte first, while the others store the low byte first. Multiple values may be used, separated by commas. If no argument is specified, then a null word is generated.

EXAMPLE:

```
DBYTE           ;generates $00,$00
DBYTE   1,2     ;generates $00,$01,$00,$02
```

DL	[LABEL]	DL	[VALUE,...]
LONG	[LABEL]	LONG	[VALUE,...]
LONGW	[LABEL]	LONGW	[VALUE,...]
LWORD	[LABEL]	LWORD	[VALUE,...]

These directives generate long word data. Values are stored with the low byte first. Multiple values may be used, separated by commas. If no values are specified, one long word of zero is generated.

EXAMPLE:

```
DL              ;generates $00,$00,$00,$00
LONG    1       ;generates $01,$00,$00,$00
LONGW   2,3     ;generates $02,$00,$00,$00,$03,$00,$00,$00
```

BLKB	[LABEL]	BLKB	NUM[,VALUE]
BLKW	[LABEL]	BLKW	NUM[,VALUE]
BLKL	[LABEL]	BLKL	NUM[,VALUE]

These directives generate a sequence of constant data. **BLKB** fills with an eight-bit value, while **BLKW** uses a sixteen-bit value and **BLKL** uses a thirty-two bit value. Values are stored low byte first. If the *VALUE* is not specified, then zero is used.

EXAMPLE:

```
BLKB 3,$FF      ;stores 3 bytes of $FF
BLKW 5          ;stores 5 words or 10 bytes of zero
BLKL            ;stores 1 long word or 4 zero bytes
```

DEFS	[LABEL]	DEFS	VALUE
DS	[LABEL]	DS	VALUE
RMB	[LABEL]	RMB	VALUE

These directives reserve the specified number of bytes in the output file. It is equivalent to storing a large number of zeros but without taking up any space in the object module file. Zeros will be generated by the linker unless the section is reference only.

EXAMPLE:

```
DEFS    0       ;don't save any space at all
DS      1       ;save one byte of memory
RMB     100H    ;save 256 bytes of memory
```

DSA	[LABEL]	DSA	VALUE
DSB	[LABEL]	DSB	VALUE
DSL	[LABEL]	DSL	VALUE
DSW	[LABEL]	DSW	VALUE

These directives reserve a specified number of bytes in the output file. It is equivalent to storing a large number of zeros but without taking up any space in the object module file. Zeros will be generated by the linker unless the section is reference only. **DSB** will generate **VALUE** bytes of space. **DSW** will generate **VALUE** times 2 bytes of space. **DSA** will generate **VALUE** times 3 bytes of space. **DSL** will generate **VALUE** times 4 bytes of space.

EXAMPLE:

```
DSB     0       ;don't save any space at all
DSW     1       ;save two bytes of memory
DSA     2       ;save six bytes of memory
DSL     1       ;save four bytes of memory
```

APWDC

This directive emulates the APW DC directive and is included for compatibility.

FLOAT	[LABEL]	FLOAT	[VALUE,...]
DOUBLE	[LABEL]	DOUBLE	[VALUE,...]

These directives generate floating point data in IEEE format. Values are stored with the low byte first. Multiple values may be used, separated by commas. If no values are specified, one long word of floating point zero is generated.

EXAMPLE:

```
FLOAT           ;generates $00,$00,$00,$00
FLOAT   1       ;generates $00,$00,$80,$3f
DOUBLE          ;generates $00,$00,$00,$00,$00,$00,$00,$00
DOUBLE  1       ;generates $00,$00,$00,$00,$00,$00,$f0,$3f
```

Macro Control

Thes directives are used to implement and control MACRO definition and execution. A complete discussion can be found in CHAPTER 5.

MACRO LABEL MACRO [ARG1,...]

This directive is used to define a macro. The LABEL is required and becomes the name of the macro. Any arguments specified will be substituted within the body of the macro. See CHAPTER 5 for more information. Macro definitions must end with a **MACEND** or **ENDM** statement.

ENDM ENDM
MACEND MACEND

These directives are used to indicate the end of the macro body. All of the macro body up to the **ENDM** or **MACEND** directive will be saved. When the macro is invoked, execution may actually terminate somewhere within the macro body by using the **MACEXIT** directive, Otherwise, execution terminates when the end of the macro body is reached. These directives can not have a LABEL.

EXAMPLE:

```
COMP  MACRO ARG1,ARG2     ;begin definition
      LDA ARG1            ;get first value
      CMP ARG2            ;compare to second
      ENDM                ;end definition
```

ARGCHK [LABEL] ARGCHK ON
[LABEL] ARGCHK OFF

This directive tells the assembler whether or not to give an error if the number of arguments passed to a macro differs from the number of arguments with which the macro was defined. The default is ON.

EXAMPLE:

```
TEST  MACRO     ARG1,ARG2     ;define dummy macro with 2 args
      NOP
      MACEND
      TEST      1,2           ;this is okay
      TEST      1             ;this will generate an error
      ARGCHK    OFF           ;turn off checking, default was ON
      TEST      1             ;now it won't
```

MACFIRST	[LABEL]	MACFIRST	ON
	[LABEL]	MACFIRST	OFF

This directive tells the assembler whether or not to generate an error when a macro definition overrides an opcode, directive or section name. When **MACFIRST** is **OFF**, an error is generated. When **MACFIRST** is **ON**, a macro can replace an opcode, directive or section name. The default is **OFF**.

EXAMPLE:

```
        MACFIRST ON
LIST    MACRO           ;dummy macro that replaces LIST
        MACEND
```

MACDELIM	[LABEL]	MACDELIM	char"7B
	[LABEL]	MACDELIM	(
	[LABEL]	MACDELIM	[

This directive specifies a delimiter character for macro arguments. Normally, a macro argument consists of all character up to but not including a comma. If you wish to include a comma within an argument, you must select an argument delimiter.

This delimiter can be used to begin an argument with it's corresponding character acting as the end of the argument. Multiple arguments must still be separated with a comma outside of any delimiters. Not all arguments to a macro need be delimited.

EXAMPLE:

```
        MACDELIM    {           ;set delimiters to {}
        COMP        {TMP,X},TMP ;using ,X requires delimiters
```

MACEXIT	[LABEL]	MACEXIT

This directive is used to terminate execution of a macro. It can occur anywhere within the macro body.

EXAMPLE:

```
COPY    MACRO   ADDR1,ADDR2   ;define COPY macro
        IF      ARG1=ARG2     ;if same address,
        MACEXIT               ; no need to copy
        ENDIF
        LDA     ADDR1         ;get value
        STA     ADDR2         ;copy it
        MACEND
```


IFMA	IFMA	ARGNUM
IFNMA	IFNMA	ARGNUM

This conditional must occur within a macro since it tests whether a given argument number exists. When a macro is invoked, the number of macro arguments passed is counted. This conditional allows the programmer to test if certain arguments are passed or not and generate the appropriate code. There is a special case if the *ARGNUM* is zero. This is a check to see if there are any arguments at all.

EXAMPLE:

```
ADD     MACRO   SRC1,SRC2,DST   ;dst=s1+s2 or s2=s1+s2
        CLC
        LDA     SRC1            ;get value
        ADC     SRC2            ;add together
        IFMA    3               ;do we have DST?
        STA     DST             ;yes, so store it
        ELSE                    ;no, so ..
        STA     SRC2            ; .. put in SRC2
        ENDIF
        ENDM

DWORDS  MACRO   VAL,NUM         ;define DWORDS macro
        IFNMA   0               ;no args?
        EXIT    DWORDS called without arguments!!
        ENDC
        DEFW    VAL,NUM
        MACEND
```

REPT	REPT	CNT
ENDR	ENDR	

This directive allows controlled repetition of assembly statements. The argument to the **REPT** directive specifies the number of times to repeat the sequence of assembly lines. The sequence is terminated by an **ENDR** directive. This directive cannot be nested.

EXAMPLE:

```
        REPT    400
        DB      $ff
        ENDR
```

Conditional Control

The directives in this section are used to perform conditional assembly. Most conditional assembly occurs within macros. More information on conditionals may be found in CHAPTER 5,which talks about macros and conditionals. All of the conditionals, the **ELSE**, **ENDC** and **ENDIF** statements can NOT have a label.

ENDC	NDC
ENDIF	ENDIF

Statements following a conditional test are parsed or skipped until an **ENDC**, **ENDIF** or **ELSE** is encountered. If an **ENDC** or **ENDIF** is encountered, then the conditional terminates.

COND	COND	EXPRESSION
IF	IF	EXPRESSION
IFFALSE	IFFALSE	EXPRESSION
IFNFALSE	IFNFALSE	EXPRESSION
IFTRUE	IFTRUE	EXPRESSION
IFNTRUE	IFNTRUE	EXPRESSION
IFZ	IFZ	EXPRESSION
IFNZ	IFNZ	EXPRESSION

These directives are all variations on the same theme. Basically, what they do is determine whether the EXPRESSION is zero or non-zero. If the expression is non-zero, then the directives **IF**, **IFNZ**, **COND**, **IFTRUE** and **IFNFALSE** will all be true with the others false. If the expression is zero, then the directives **IFZ**, **IFFALSE** and **IFNTRUE** will be true with the first set false.
EXAMPLE:

```
IFNZ    SIZE    ;do we have any space to save?
RMB     SIZE    ;yes, save the space
ENDIF
```

ELSE	ELSE

Statements following a conditional test are parsed or skipped until an **ENDC**, **ENDIF** or **ELSE** is encountered. If an **ELSE** is encountered, then the following statements are skipped or parsed until an **ENDC** or **ENDIF** is encountered. Any **IF** ..... **ELSE** ..... **ENDIF** sets that are nested will be skipped as well.

EXAMPLE:

```
IF      SIZE>2      ;bigger than a word?
LONG    0           ;yes, define long value
ELSE
WORD    0           ;no, define word value
ENDIF
```

NOTE: No spaces allowed in the conditional equation (i.e., size>2)

IFABS	IFABS	LABEL
IFNABS	IFNABS	LABEL
IFREL	IFREL	LABEL
IFNREL	IFNREL	LABEL

These directives are used to determine if a particular symbol has been defined as absolute or relative. Labels are created two ways. The first way is to mark the current location of an opcode or directive within a section. If the section within which the label is located is ABSOLUTE, then the label is ABSOLUTE. If the section is RELATIVE, then the label is RELATIVE. The second way symbols are defined is through the **EQU** directive. If the operand of the **EQU** directive is ABSOLUTE, then the label is absolute. If the operand is RELATIVE, then the label is RELATIVE. If the symbol is not in the symbol table or is undefined, an error is registered.

EXAMPLE:

```
LAB1    EQU     3               ;constant is ABSOLUTE
LAB2    EQU     EDATA-BDATA     ;label-label is always ABSOLUTE
LAB3    EQU     BDATA+3         ;if BDATA is RELATIVE, so is LAB3

        IFABS   LAB1            ;true
        IFNABS  LAB2            ;false
        IFREL   LAB3            ;true if BDATA is RELATIVE
        IFNREL  LAB1            ;true
```

IFDEF	IFDEF	LABEL
IFNDEF	IFNDEF	LABEL

The symbol table is searched for the specified symbol name. If found, the symbol is considered defined. Symbols are defined by being used as a label within a section or by using the **EQU** or **SET** directives.

EXAMPLE:

```
DEBUG   EQU     1               ;turn on debugging

        IFDEF   DEBUG           ;is debugging on?
        PEA     #MESSAGE        ;yes, so push message
        JSR     PRINT           ;and print it
        ENDIF
```

IFDIFF	IFDIFF	STR1,STR2
IFNDIFF	IFNDIFF	STR1,STR2
IFSAME	IFSAME	STR1,STR2
IFNSAME	IFNSAME	STR1,STR2

This conditional compares the two strings that are the arguments. Case IS significant, which is to say that `a' is not the same as `A'. The strings may not contain white space unless enclosed with single or double quotes. If the two strings are identical, then **IFSAME** and **IFNDIFF** will be true and **IFDIFF** and **IFNSAME** will be false. This directive is most often used inside of a macro.

EXAMPLE:

```
CHK     MACRO       WHICH,VALUE     ;define CHK macro
        IFSAME      NAME,word       ;if comparing a word
        CMP         #VALUE          ;just do it
        ELSE                        ;otherwise
        SEP         #$20            ;go to 8 bit mode
        LONGA       OFF             ;tell assembler
        CMP         #VALUE          ;do the compare
        REP         #$20            ;back to 16 bit mode
        LONGA       ON
        ENDIF
        MACEND

        CHK         word,0
        CHK         byte,4
```

IFEXT	IFEXT	LABEL
IFNEXT	IFNEXT	LABEL

The specified symbol is checked to see if has been marked as external. The directives **GLOBAL**, **PUBLIC**, **XDEF**, **EXTERN**, **EXTERNAL** and **XREF** are used to mark a symbol as external. The **GLOBALS** directive is used to mark all symbols as external or not. An error is generated if the symbol is not defined.

EXAMPLE:

```
LAB1:
        XREF        LAB1
LAB2:
        IFEXT       LAB1        ;true
        IFEXT       LAB2        ;false
        IFNEXT      LAB2        ;true
```

IFPAGE0	IFPAGE0	LABEL
IFNPAGE0	IFNPAGE0	LABEL

This conditional is true if the symbol specified has been defined in a **PAGE0** section. Symbols defined in a **PAGE0** section are used for direct page addressing in the 65816 core parts. This conditional provides a mechanism to generate two different sequences of code depending on whether the symbol is located in the direct page. If the symbol is not in the symbol table, an error is generated. I.e. for the 6502, it means address $00-$FF. For the 65816, it means address $00:0000 - $00:FFFF.

EXAMPLE:

```
GETVAL  MACRO   ADDR    ;define GETVAL macro
        IFPAGE0 ADDR    ;is symbol direct page?
        LDA     (ADDR)  ;yes, use it
        ELSE
        LDA     ADDR    ;no, so copy value
        STA     <0      ; to the direct page
        LDA     (0)     ;and use it there
        ENDIF
        MACEND
```

IFLONGA IFLONGA
IFLONGI IFLONGI
IFSHORTA IFSHORTA
IFSHORTI IFSHORTI

This conditional tests the state generated by the **LONGA** and **LONGI** directives.

EXAMPLE:

```
        IFLONGA
        LONGA   OFF
                        ENDIF
```

IFMATCH IFMATCH STR1,STR2,CNT

This conditional compares two strings specified by *STR1* and *STR2* for *CNT* characters.

EXAMPLE:

```
LOAD    MACRO   ARG,VAL
        IFMATCH ARG,"R0",2
        LDA     #VAL
        ELSE
        LDA     #>VAL
        ENDIF
        ENDM

        LOAD    R0,3
```

Listing Control

The directives in this section control the appearance of the listing file generated with the **-L** option to WDCxAS.

PL [LABEl] PL VALUE

This directive sets the page length for the listing file. The default is **61** but the first **PL** directive encountered during Pass 1 will be used for the first page.
EXAMPLE:

```
        PL      66      ;set my page length
```

PW	[LABEl]	PW	VALUE

This directive sets the page width for the listing file. The default is **78** but the first **PW** directive encountered during Pass 1 will be used for the first page.

EXAMPLE:

```
PW      132     ;use a wide carriage printer
```

TOP	[LABEL]	TOP	VALUE

This directive specifies the number of blank lines to be printed at the top of each page before the date and page number line. The default is **0**, but the first **TOP** directive encountered during Pass 1 will override the default for the first page.

EXAMPLE:

```
TOP     5       ;leave some room at top for binding
```

HEADING	[LABEl]	HEADING	TEXT
NAM	[LABEl]	NAM	TEXT
TITLE	[LABEl]	TITLE	TEXT
TTL	[LABEl]	TTL	TEXT

This directive is used to indicate the text of the title that is to be printed at the top of each page. If none of these directives is ever encountered, then there will be no title line printed. The first of these directives detected during Pass 1 will be saved and used to title the first page. If you do not wish a title on the first page, use a an empty **HEADING** directive before the first real **HEADING** directive. The first two blanks or tabs following the directive will be skipped. Any remaining blanks or tabs will be used to print the title. This allows the title to be centered.

EXAMPLE:

```
HEADING     This is my heading ; this is part of it, too!
TITLE               Centered heading.
```

STTL	[LABEl]	STTL	TEXT
SUBTITLE	[LABEl]	SUBTITLE	TEXT

SUBTTL	[LABEl]	SUBTTL	TEXT

This directive is used to indicate the text of the subtitle that is to be printed at the top of each page below the title line. If none of these directives is ever encountered, then there will be no subtitle line printed. The first of these directives detected during Pass 1 will be saved and used to subtitle the first page. If you do not wish a subtitle on the first page, use a an empty SUBTITLE directive before the first real SUBTITLE directive. The first two blanks or tabs following the directive will be skipped. Any remaining blanks or tabs will be used to print the subtitle. This allows the subtitle to be centered.

EXAMPLE:

```
SUBTITLE    This is my subtitle ; this is part of it, too!
STTL                Indented subtitle
```

EJECT	[LABEl]	EJECT	
PAG	[LABEl]	PAG	
PAGE	[LABEl]	PAGE	

This directive outputs a form-feed to the listing file causing a new page to be started.

EXAMPLE:

```
ENDS        ;end of previous section
EJECT       ;start new section on new page
DATA        ;start new section
```

LIST	[LABEl]	LIST	ON
NLIST	[LABEl]	LIST	OFF
NOLIST	[LABEl]	NLIST	
	[LABEl]	NOLIST	

This directive controls the listing of assembly language statements to the listing file. If the **-L** option has not been specified, no listing file is produced. This can be used to suppress the listing of **INCLUDE** files or other sets of data statements. See the following **INCLIST** directive as well. The default is **ON**.

EXAMPLE:

```
NOLIST                  ;no macros in listing
INCLUDE     MACROS.INC  ;parse the macros
LIST        ON          ;turn listing back on
```

INCLIST	[LABEl]	INCLIST	ON
	[LABEl]	INCLIST	OFF

This directive controls whether include files are listed in the listing file. The default is **ON**.

EXAMPLE:

```
INCLIST     OFF         ;no macros in listing
INCLUDE     MACROS.INC  ;parse the macros
                        ;remaining lines will list
```

ASCLIST	[LABEl]	ASCLIST	ON
	[LABEl]	ASCLIST	OFF

This is one of the directives that control how listings are generated. When a listing is created, the actual bytes that are generated for instructions and directives are output as hexadecimal values. When **ASCLIST** is **ON**, assembly directives that generate more than four bytes of data will display the additional values on as many additional lines as are necessary. When **ASCLIST** is **OFF**, only the first line of values is displayed. All values will be generated into the output file, only the display is truncated. The default is **ON**.

EXAMPLE:

```
1       00:0000: 61 62 63 64    DB              'abcdefg'
        00:0004: 65 66 67
2                               ASCLIST OFF
3       00:0007: 61 62 63 64    DB              'abcdefg'
4       00:000E: EA             NOP
```

CONDLIST	[LABEl]	CONDLIST	ON
	[LABEl]	CONDLIST	OFF

This directive controls the printing of conditionals that are not executed. For example, if an **IF** statement is false, any lines up to an **ELSE** or **ENDIF** or **ENDC** directive are not parsed by the assembler. If **CONDLIST** is **ON**, then these lines will be printed to the listing file if listing is activated. If **CONDLIST** is **OFF**, then these lines are not printed. When the conditional is true, then the lines are always printed. The default is **ON**.

EXAMPLE:

```
1                       IF          0
2                       NOP
3                       ELSE
4 00:0000: 60           RTS
5                       ENDIF
6
7                       CONDLIST    OFF
8                       IF          0
10                      ELSE
11 00:0001: 60          RTS
12                      ENDIF
```

MACLIST	[LABEl]	MACLIST	ON
MLIST	[LABEl]	MACLIST	OFF
MNLIST	[LABEl]	MLIST	
	[LABEl]	MNLIST	

This directive controls the listing of expanded macros. When a macro is invoked, the invoking line is displayed, followed by each line of the macro as it is parsed by the assembler. Each expanded macro line is precede by a `+' symbol. If the macro exits with a **MACEXIT** directive, the remaining lines of the macro are not listed. The default is **ON**.

EXAMPLE:

```
          1                          NULL        MACRO       SIZE
          2                          IF          SIZE=2
          3                          DW          0
          4                          ELSE
          5                          DL      0
          6                          ENDIF
          7                          MACEND
          8
          9                          NULL        2
+         9                          IF          2=2
+         9 00:0000: 00 00           DW          0
+         9                          ELSE
+         9                          DL          0
+         9                          ENDIF
          10                         MACLIST OFF
          11 00:0002: 00 00 00 00    NULL        4
```

PASS1	[LABEl]	PASS1	ON
	[LABEl]	PASS1	OFF

This directive controls whether a listing is generated during Pass 1. This is useful for checking macros and tracking down syntax errors. The default is **OFF**.

EXAMPLE:

```
          1                          PASS1   ON
          2            xxxxxxxx      BB      EQU     CC
          3 00:0000:                 DB       BB
          4            00000001      CC      EQU     1

          1                          PASS1   ON
          2            00000001      BB      EQU     CC
          3 00:0000: 01              DB       BB
          4            00000001      CC      EQU     1
```

Appendix A Assembler Error Messages

Fatal Errors

Premature end of file in conditional.

After a conditional directive is encountered such as .IF, if the end of the file is encountered before an end conditional is encountered it is considered a fatal error.

Modules must start and end in original file!

A MODULE or ENDMOD directive has been encountered within an included file.

Unable to start new module without ENDMOD.

A new MODULE directive has been encountered before the old module was terminated with an ENDMOD directive.

Need module name here.

A new MODULE directive has been detected without a name for the module immediately following it.

More than one input file specified!

The assembler only assembles a single file at a time. If more than one input file name is specified, this error is generated.

More than one output name.

Only one output or list file can be specified.

Out of memory!

The assembler was unable to allocate memory for an operation.

No input file specified!

The assembler needs an input file name to be passed as an argument when the assembler is invoked.

Can't open input file <FILE>.

The assembler was unable to open the file, FILE, for reading.

Can't open output file <FILE>.

The assembler was unable to open the output file, FILE, for writing.

Can't open listing file <FILE>.

The assembler was unable to open the listing file, FILE, for writing.

Too many -I options.

The assembler allows for at most 16 -I include path options.

Includes nested too deep.

The assembler allows include files to be nested at most fifty deep.

Unable to reopen 'FILE' after INCLUDE!

The assembler only opens one file at a time when handling include files. After it finishes a file, it reopens the previous file and seeks to where it left off. If the previous file can't be opened, this error is generated.

Input line longer than 512 characters!

The assembler only allows input lines up to 512 characters in length.

Missing MACEND or ENDM in macro definition.

If an end-of-file is encountered during a macro definition before the macro definition is terminated by a MACEND or ENDM directive, this error is generated.

Macro nested more than 256 deep!

Macros may be nested at most 256 deep in the assembler.

Macro arguments too long!

Macro arguments may be up to 128 characters in length.

Reference to undefined macro argument!

If a reference is made to a macro argument that is not defined as an argument to the macro, it is an error.

Expanded macro line longer than 512 characters!

After macro expansion, the resulting line may not be longer than 512 characters.

REPT line longer than 512 characters!

A REPT directive may not expand to more than 512 characters.

Missing ENDREPT in REPT definition.

If an end-of-file is encountered during a REPT definition before the REPT definition is terminated by an ENDREPT directive, this error is generated.

Error writing to object file.

If an error occurs while writing to the output file, this error is generated.

Label value different between pass 1 and 2!

This is an internal error which is generated when a label value has a different value on pass 2 than on pass1. This is a safety check that nothing changes size between pass 1 and pass 2.

Error writing to listing file.

This is generated if writing to the listing file returns an error.

Exceeded maximum of 256 sections!

The assembler supports a maximum of 256 different sections.

Max of 500 nested sections exceeded!

The assembler allows nested section directives, but only up to 500.

Imbalance in nested sections.

If an ENDS directive is encountered without a corresponding section directive, this error is generated.

Non-Fatal Errors

Need symbol name here!

A symbol name must follow an IFDEF or IFNDEF directive.

Missing comma and second argument.

The IFDIFF, IFSAME, IFNDIFF and IFNSAME directives need two arguments separated by a comma.

Conditional requires symbol name.

The IFEXT, IFABS, IFREL, IFPAGE0, IFNEXT, IFNABS, IFNREL, IFNPAGE0 directives must be followed by a symbol name.

Unknown symbol in conditional.

The IFEXT, IFABS, IFREL, IFPAGE0, IFNEXT, IFNABS, IFNREL, IFNPAGE0 directives must be followed by a defined symbol name.

This conditional only valid inside a macro.

The IFMA and IFNMA directives are only valid within the body of a macro definition.

Need start,size for INSERT!

The INSERT directive must be followed by the name of the file to insert, the location to insert it and the size.

Couldn't open binary file 'FILE'!

The INSERT directive was unable to open the file specified for insertion.

Symbol required.

The EXTERN, EXTERNAL, XREF, GLOBAL, PUBLIC, and XDEF directives must be followed by at least one symbol and only by symbols.

Label is required for directive.

The EQU, EQUAL, GEQU, DEFL, SET and VAR directives must be preceded by a label.

Label type redefined.

The DEFL, SET and VAR directives may not redefine a label previously defined.

Can't redefine type of label.

The EQU, EQUAL and GEQU directives may not redefine a lable previously defined.

Fully resolved expression required for EQU by Pass 2!

An EQU directive may contain forward references, but they must be resolved by the end of the file.

Too many global equates.

The assembler only allows up to 1000 global equates.

Page length must be at least 10 lines!

Attempting to change the page length using the PL directive will not allow lines less than ten lines.

Page width must be >= 40 and <= 132!

The PW directive may change the width of the output page to between 40 and 132 characters only.

Too many lines on top!

The TOP directive may not specify more lines than are on the output page.

Missing termination character 'X'!

The FCC directive uses the first character of it's argument as the terminator and looks for that character to terminate the argument.

Illegal outside of macro definition!

The ENDM and MACEND directives are only valid within the definition of a macro.

Illegal outside of rept definition!

The ENDREPT directive is only valid within the definition of a REPT.

Only valid delimiters are: {, (, and [.

The MACDELIM directive allows specifying the character used to delimit a macro argument, but is limited to the characters '{', '(' and '['.

MACEXIT illegal outside of macro definition!

The MACEXIT directive is only valid within the definition of a macro.

Conditional ELSEIF directive out of place.

The ELSEIF directive is only valid after a conditional directive.

Need conditional end directive here.

An end of condidional directive must be seen before the end-of-file is reached.

Conditional ELSE directive out of place.

The ELSE directive is only valid after a conditional directive.

Conditional end directive out of place.

An end of condition directive is only valid after a conditional directive.

Couldn't find section during pass2!

This error occurs if a section directive is parsed on pass 2 that wasn't parsed on pass 1.

Label is required for SECTION directive.

A label must be on the same line as a SECTION directive to name the section.

Illegal value for RADIX directive!

The RADIX directive only allows the values 1, 2, 8, b, d, h, o, q. Anything else is illegal.

Need CHIP type here!

The CHIP directive must be followed by a symbol or number.

Invalid CHIP type!

The CHIP directive must be followed by one of "6502", "65c02", "w65c02s", "65816", or "65802". Anything else is invalid.

End marker for comment missing!

The COMMENT directive takes a character to be used as an end marker as its argument. This error is generated if the character is missing.

End of file before end of comment!

This error is generated if the end-of-file is reached before the end of comment marker from a COMMENT directive is detected.

Need character argument for directive!

The LLCHAR directive is used to change the special character used to denote temporary labels. This error message is generated if the LLCHAR directive is give without an argument.

Need quoted file name!

The FILE directive takes a file path surrounded by double quotes followed by a comma and a line number. If the quotes are missing, it is an error.

Need line number after file name arg.

The FILE directive takes a file path surrounded by double quotes followed by a comma and a line number. If the comma and/or line number are missing, it is an error.

Need local offset in ENDFUNC directive.

The ENDFUNC directive takes three arguments separated by commas. If the second argument, the local offset, is missing, it is an error.

Need arg offset in ENDFUNC directive.

The ENDFUNC directive takes three arguments separated by commas. If the third argument, the argument offset, is missing, it is an error.

Symbol required.

The SYM, STAG and MEMBER directives require a symbol as the first argument to the directive.

Symbol value required in SYM/MEMBER directive!

Following the symbol name, in the SYM and MEMBER directives is the value of the symbol.

Unimplemented assembler directive.

If an opcode is not a valid opcode and not a directive, it is considered to be an unimplemented directive.

Missing argument.

A directive that takes an ON or OFF argument is missing the argument.

Bad argument.

A directive that takes an ON or OFF argument has something other than ON or OFF as the argument.

Divide by zero!

While evaluating an expression, a divide or modulo by zero was detected.

Invalid operator in floating point evaluate - N.

While evaluating a floating point expression, an invalid operator such as shift was detected.

Bank number out of range!

An address expression contains a bank number larger than 255.

Hex and symbol are identical!

A hex number matches a symbol name. Use a leading zero to avoid this error.

String not terminated!

While parsing the input, a string was detected that was not terminated by the corresponding termination character.

Missing terminating '.' on operator!

Operators that require a leading and trailing period such as .UGT., require both the leading and the trailing period.

Macro name already defined!

A macro name may only be defined once. Attempting to define it twice is an error.

Macro name conflicts with opcode, directive or section!

A macro name may not be the same as an opcode, directive or section unless the MACFIRST directive has been turned on.

Arguments must be valid names.

When a macro is defined, the arguments must be valid symbol names.

Different number of arguments in macro call(N) and definition(N).

When a macro is invoked, the number of arguments in the call should match the number of arguments in the definition.

Too many global equates.

The assembler only allows up to 1000 global equates.

Illegal index register!

Only 's', 'x', and 'y' are valid as index registers.

Missing character!

When an argument starts with a '(' or '[', there should be a matching ')' or ']'.

Only index register indirect allowed!

Only the Y index register is allowed following a stack addressing mode.

Only Y index register allowed!

Only the Y index register is allowed following a close parenthesis.

Illegal addressing mode!

The addressing mode specified is not a legal addressing mode.

Can't use register as label!

A register name may not be used as a label.

Need symbol after '.'!
Need trailing '.'!
Only #.low. or #.high. allowed!

When using immediate addressing, a .low. or .high. may be used to select the byte to be used. When a period follows the '#' character, it is assumed that a symbol will follow followed by another period and that the symbol will be one of "low" or "high".

Instruction not allowed with selected processor.
Addressing mode not allowed with selected processor.

Different processors support different sets of instructions and addressing modes. The processor is selected with the CHIP directive. If an instruction not implemented on the selected processor is used, this error is generated.

Immediate value truncated!

When an immediate value is used in such a way that the value is changed when the value is truncated, this error message is given.

Need label to branch to!

Branch instructions require a label to branch to.

Branch out of range!

Branches are limited to plus or minus 127 bytes. If a branch is attempted to a label beyond this range, an error is generated.

Dot not allowed on opcode names.

A leading period is allowed on directive names, but is not allowed on instruction opcodes.

Multiply defined symbol.

An attempt has been made to define a symbol that has already been defined.

Illegal character in directive.

A leading period must be followed by a directive otherwise it is an error.

Need opcode, directive or macro name here.

A statement must consist of either an opcode, directive or macro name.

Unknown opcode, directive, macro or section.

A statement has a symbol that is not defined.

Extra characters on line!

After the last argument, the remainder of the line should be empty except for comments unless the SPACES directive has been used.

Section name already defined!

An attempt has been made to define a section already defined.

Section name conflicts with opcode, directive or macro!

A section name must be unique and not conflict with any opcode, directive or macro.

Undefined symbol - <SYM>.

A symbol has been referenced but not defined.

Appendix B Linker Error Messages

Error creating symbol file!

An error occurred while trying to create the symbol file.

Too many source files in module!

The linker allows up to 5000 different source file names. More than that is an error.

Unable to find tag serial number!

This is an internal error that occurs when dealing with structure tags.

Couldn't create error file `FILE'!

An error occurred while trying to create the error file.

Error creating symbol listing file!

An error occurred while trying to create the map file.

Linkio:Out of memory!

The linker output cache mechanism was unable to allocate enough memory.

Cannot create output file: ZLN.TMP!

The linker creates a temporary file when building the output. This error occurs if the linker is unable to create that file.

Error while lseeking output file!

An error occurred when seeking in the temporary output file.

Error writing output file!

An error occurred when the cache is written to the output file.

Error while reading output file!

An error occurred when a cache block was read back from the output file.

Attempted to write outside of file bounds!

The linker pre-computes where everything in the output should end up, and if something ends up outside the computed bounds of the output file it is an internal error and should be reported.

No input given!

When the arguments to the linker are parsed, if no input files have been specified, that is considered an error.

Option syntax error!

When an argument is passed to the linker, if there are extra characters that are part of the option, it is an error.

Cannot have nested -f options.

The -f argument allows options to be read from a text file. However, the option files may not be nested.

Cannot open -f file: FILE!

The linker was unable to open the options file, FILE.

Illegal Nintendo map!

The Nintendo map option may be specified as -n, -n2, or -n8. Anything else is an error.

Out of memory!

The linker is unable to allocate enough memory.

Couldn't open FILE in pass 2!

The linker was unable to open a module file in pass 2 that was opened during pass 1.

Unknown loader item (0xXX)!

An object module is corrupt and the linker has detected an unknown token.

Section 'SECT' has different type in module 'FILE:MODULE'!

The same section can be defined in different files. However, all definitions of the same section should be defined with the same parameters, otherwise it is an error.

Overlap of NN bytes in section 'SECT' of 'FILE:MODULE'!

The linker checks that sections located at fixed addresses don't overlap one another.

Section SECT's ROM image exceeds bank $XX by $XX!

A single section may not exceed 64K unless the section spread option, -Z, has been specified.

Attempt to locate section `SECT' more than once!

A section may be located at only a single fixed address.

Module 'FILE:MODULE' too big to fit!

As each module's data is added to a section, if there is not enough room left in the bank, this error is generated.

Section 'SECT' overlaps section 'SECT' by NN bytes at address 0xXX (ROM)
Section 'SECT' overlaps section 'SECT' by NN bytes at address 0xXX (Relocatable)

Each section's ROM and relocatable address and size is checked to see if it overlaps another section and this message is generated if it does.

Can't mix 65xxx and 65032 object module types!

The object module format for 6502 and 65816 is different from the 65032.

Library format is invalid!

If a file is passed to the linker as a library and is not in the correct format, an error is generated.

Can't open FILE!

The specified file, FILE, didn't exist or was locked and couldn't be opened.

Couldn't read object file FILE!

An error occurred when reading the specified file, FILE.

Not an object file FILE!

The object file is not in the correct format.

Undefined symbol: SYM

A symbol is referenced that has not been defined.

Branch out of range!

The linker can handle files assembled in a single pass and can resolve branch relative instructions and generates an error if the branch is out of range.

THIS PAGE LEFT INTENTIONALLY BLANK

INDEX

