MPE VFX C

For the RTX 2000

User Guide

MicroProcessor Engineering Limited
Rowley Associates Limited

Second edition.

Rowley Associates Limited
8 Silver Street
Dursley
Gloucestershire GL11 4ND
UNITED KINGDOM

Tel:+44 1453 547916
Fax: +44 1453 544068

e-mail: info@rowley.co.uk

MicroProcessor Engineering Limited
133 Hill Lane
Shirley
Southampton SO15 5AF
UNITED KINGDOM

Tel: +44 1703 631441
Fax: +44 1703 339691

email: support@mpeltd.demon.co.uk

Contents

1 Introduction 7

Welcome! 7
Quick start 7
Text conventions 7
Standard syntactic metalanguage 8
Further reading 9

2 RTX-2010 Features 11

Introduction 11
Memory architecture 11
ROMs, sections, and initialisation 12
Processor startup 14
Data representation 14
Compiling code to separate sections 15
Code quality issues 15
Calling conventions 16

3 Compiling Files 19

Introduction 19
Naming conventions 19
Translating files 20
A rough guide to assembler options 21

4 Source Structure 23

Introduction 23
Writing assembly language statements 23
Constants 24
Comments 26

5 Labels and Variables 29

Introduction 29
Label names 29
Symbolic constants or equates 29
Using type specifiers 30
Labels 30
Defining and initialising data 31
Aligning data 32
Filling areas 33

6 Data Types 35

Introduction 35
Built-in types 36
Structure types 36
Array types 38
Pointer types 39

7 Expressions 41

Introduction 41
A note on operators and operands 41
Byte and word extraction operators 41
Index operator 42
Bitwise operators 42
Arithmetic operators 43
Relational operators 44
Logical operators 45
Miscellaneous operators 46

8 Sections 49

Introduction 49
Selecting a section 49
Predefined sections 50

9 Conditional Assembly 51

Introduction 51
General structure 51
Cascading conditionals 52
Typical uses 52

10 Using Multiple Modules 55

Introduction 55
Exporting symbols 55
Importing symbols 56
Other terminology for import and export 56
Libraries 57

11 RTX Assembler Reference 59

Introduction 59
Shifter instructions 59
ALU operations 61
Instructions 63

12 Compiler Driver Reference 69

Introduction 69
Command syntax 69
Verbose execution 71
Where to find included files 71
Setting the output format 71
Macro definitions 72
Linking in libraries 73

13 The Linker 75

Introduction 75
Linker function and features 75
Command syntax 76
Linker output formats 77
Laying out memory 77
Linkage maps 78
Linking in libraries 78

14 The Archiver 79

Introduction 79
Command syntax 79
Archive operations 80

15 The Object File Lister 83

Introduction 83
Command syntax 83
Listing section attributes 84

16 The Hex Extractor 85

Introduction 85
Command syntax 85
Supported output formats 86
Preparing images for download 86
Preparing images for ROM 87
Extracting parts of applications 87

Index 89

1

Introduction

Welcome!

Welcome to VFX C for RTX 2000! VFX C allows you prepare C applications which run embedded on the Harris RTX 2000 family processors.

This introductory chapter covers the contents of and conventions used in this manual: it should be read by everyone, but...

Quick start

If you can't wait to install and try out the VFX software, simply plug in the CD ROM and the installer should run automatically. If it doesn't you may have insert auto-notification turned off, so explore the CD and run the setup program in the CD's root directory.

If you're migrating from the Harris or Metsys C tools, you will want to look at the Migration section which will give you a synopsis of the changes you can expect with VFX C.

Text conventions

Throughout this manual, text printed in `this typeface` represents verbatim communication with the computer: for example, pieces of C text, commands to the operating system, or responses from the computer.

In examples, text printed in *this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of one kind of compilation command:

`hcl` ***source-file***

This means that the command consists of:

- The word '`hcl`', typed exactly like that.
- A *source-file*: not the text source-file, but an item of the *source-file* class, for example '`myprog.c`'.

Whenever commands to and responses from the computer are mixed in the same example, the commands (i.e. the items which you enter) will be presented in *`this typeface`*. For example, here is a dialogue with the computer using the format of the compilation command given above:

```
venera% hcl -v myprog.c
VFX Compiler Driver   Release 1.0.0
Copyright (c) 1999 MicroProcessor Engineering limited
Copyright (c) 1997, 1998, 1999 Rowley Associates Limited.
```

The user types the text '`hcl -v myprog.c`', and then presses the enter key (which is assumed and is not shown); the computer responds with the rest.

When numbers are used in the text they will usually be decimal. When we wish to make clear the base of a number, the base is used as a subscript, for example 15_8 is the number 15 in base eight and 13 in decimal, $2F_{16}$ is the number 2F in hexadecimal and 47 in decimal.

Standard syntactic metalanguage

In a formal description of a computer language, it is often convenient to use a more precise language than English. This language-description language is referred to as a *metalanguage*. The metalanguage which will be used to describe the C language is that specified by British Standard 6154. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory.

The BS6154 standard syntactic metalanguage is similar in concept to many other metalanguages, particularly those of the well-known Backus-Naur family. It therefore suffices to give a very brief informal description here of the main points of BS6154; for more detail, the standard itself should be consulted.

- Terminal strings of the language—those built up by rules of the language— are enclosed in quotation marks.
- Non-terminal phrases are identified by names, which may consist of several words.
- A sequence of items may be built up by connecting the components with commas.
- Alternatives are separated by vertical bars ('`|`').
- Optional sequences are enclosed in square brackets ('[' and ']').
- Sequences which may be repeated zero or more times are enclosed in braces ('`{`' and '`}`').

- Each phrase definition is built up using an equals sign to separate the two sides, and a semicolon to terminate the right hand side.

Further reading

This user manual does not attempt to teach the C language itself; rather, reference should be made to one of the many introductory texts available.

The reader is assumed to be fairly familiar with the operating system of the host computer being used. For Microsoft Windows development environment we recommend Windows NT 4, but you can use Windows 95 or Windows 98 if you wish.

2

RTX-2010 Features

Introduction

This section describes the specific features which apply to the MPE C compiler for the Harris RTX 2010.

Memory architecture

The VFX C compiler supports code generation for two memory models, large model and small model. In small model, code and data are both limited to 64 kilobytes and each must be completely contained within one 64 kilobyte page. In large model, code and data are unrestricted and can be placed into multiple 64 kilobyte pages.

The architecture of the RTX 2010, however, imposes a number of restrictions on the way code and data are allocated to memory in large model.

Byte ordering

The compiler must know the byte order for data in advance so that it can correctly initialise byte-oriented data, such as string constants, in memory. As such, the compiler and runtime system *require* that the processor be run in big-endian mode where the most significant part of a word is at the lower address.

At present the compiler does not assume any ordering for byte-ordered data in the frame, leading to suboptimal code. As a future enhancement, the compiler may well take advantage of knowing byte ordering in external memory to optimize access to user memory.

Frame access and placement

In small model the C stack frame and all static data must be placed into the same 64 kilobyte page. Using this assumption, the compiler dispenses with all page-switching code during normal operation.

In large model the C stack frame and static data can be placed into separate pages. In this mode, the C compiler arranges for the correct page number to be placed into the data page register immediately before the access to the paged data.

Data object restrictions

The compiler and runtime system *require* that no object spans a page boundary. Allowing this would seriously damage the performance of both compiled code and the runtime system. In effect, the compiler and runtime system generate code first to select the appropriate page and then to operate on an object *completely within that page*.

Code restrictions

Code must be placed so that it does not flow across page boundaries as the RTX processor does not correctly support execution across page boundaries. In large model, code can call routines in any page and access data in any page; there in no requirement to batch common routines together and allocate them to the same physical page.

ROMs, sections, and initialisation

If you intend to deploy your application in ROM you will need to take special care that the appropriate sections are initialised correctly when the runtime system starts. This means that the `.bss` section must be cleared and the `.data` section copied from ROM to RAM.

The linker provides the necessary options to separate the place where data are located at runtime and the place where they are burnt into EPROM. The place where data are located at runtime is their *run address*, and the place where a copy of the initialised data are kept ready to be copied is called their *load address*.

Setting up uninitialised data areas

The `.bss` section is used to hold uninitialised data which must be zeroed before starting the main program. You can zero the `.bss` in the runtime startup by using the special linker-created symbols which define the extent of sections.

The lowest run address of the .bss section is named .bss$$start and the highest is .bss$$end, which is the byte immediately following the end address of the section.

Initialisation of .bss could be coded as follows:

```
    LIT    HWORD(.bss$$start)    ; page which contains bss data
    GSTORE 13                    ; select bss page
    LIT    .bss$$end-.bss$$start ; size of .bss section
    BZP    Done                  ; zero size, nothing to do
    LIT    1
    SUB                          ; loop cound
    TOR
    LIT    LWORD(.bss$$start)    ; in-page address of bss
Init
    DUP
    LIT    0
    CSTORE
    LIT       1
    ADD
    BINZ      Init
    DROP
Done
```

This zeroes the complete .bss section. You can use the same prototype code to initialise other sections which you have defined

Setting up initialised data areas

Writable initialised data must be set up before calling the main routine. Obviously, if initialised data is placed into volatile memory, we need a mechanism to keep an image of the initialised data in permanent storage so that the working store can be set up.

You can do this by directing the linker to place an image of the initialised data at an address in ROM, but to treat the data as if they were placed at another address in RAM.

Consider a system where RAM extends from 0 through $FFFF_{16}$, and ROM from 18000_{16} through $1FFFF_{16}$. Suppose we wish to place our data in RAM at address 200_{16}, and code and initialisation tables in ROM at 18000_{16}. We ask the linker to do this using the option -T.data=18000/200. This instructs the linker to place the .data section at address 18000_{16} but to make all references to items in that area use addresses 200_{16} and up.

The problem is to now transfer the data from ROM to RAM on startup. The linker provides four symbols for this, .data$$start and .data$$end define the range of the runtime data section just as we saw with the .bss section, and similarly .data$$load$$start and .data$$load$$end define the extent of the image of the data section in ROM.

We copy form ROM to RAM using the ___lbcopy routine of the runtime library:

```
LIT   LWORD(.bss$$start)       ; run (RAM) address
LIT   HWORD(.bss$$start)
LIT   LWORD(.bss$$load$$start) ; load (ROM) address
LIT   HWORD(.bss$$load$$start)
LIT   .bss$$end-.bss$$start    ; size
CALL  ___lbcopy
```

Processor startup

We provide a set of startup routines for the MPE RTX-2000 PowerBoard, which is the reference platform the compiler and tools are tested. This board provides up to 512K of EPROM and 512K of RAM that can be used to store code and data. However, you will need to modify these for your target before any code will run on it.

Building the startup code

The startup code defines the entry point `start` and initialisation for the on-board devices. There is a section which is used for the processor stack, called '`.stack`' and is initially one kilobyte in size. You can build the demo program with this runtime startup code using the following command line:

```
hcl crt0.s demo.c -T.stack=10000 -T.text=200 -T.data=1000 -Ns
```

This compiles the startup code together with the demo, does not link in the standard startup code `lib/crt0.hzo`, places the stack at address 0 in page 1, and places the code at address 200_{16} in page 0..

You will need to customize the startup code for your board if it differs significantly from the MPE board.

Installing the startup code

Once you have built the startup code for your board, you should place it in the `lib` directory with the name `crt0.hzo`. When you omit the `-Ns` switch from the command line, the compiler driver automatically links the `crt0.hzo` startup code into your application.

Data representation

Long integers

Long integers are held in memory in big-endian word order. Thus, the most significant part of the long is in the word at the lower address.

When a long value is loaded onto the stack, the high 16 bits of the value are stored above the low 16 bits.

Paged pointers

In large model, pointers are 32 bits wide. In this case, the low 16 bits of the pointer are the offset within the page, and the high 16 bits select the page in the RTX address space (only the low four bits are required to select the page and the compiler and linker zero the undefined bits).

All increments of paged pointers

Compiling code to separate sections

The VFX C compiler allows code to be generated into any named section on a global basis, whereas the Horizon C compiler supports allocation of code on a per-function and per-global basis. Because the two compilers differ, the following will describe how to place data using the VFX C compiler provided with this product.

By default, the VFX C compiler generates code into `.text`, read-only constants into `.rodata`, initialised data into `.data`, and uninitialised data into `.bss`. You can, however, change the section names the compiler uses with the `-B` option:

Option	Description
`-Bt`*name*	Generate code into section *name*
`-Bd`*name*	Generate initialised data into section *name*
`-Bb`*name*	Generate uninitialised data into section *name*
`-Bc`*name*	Generate read-only constants into section *name*

Table 2-1 **Setting compiler section names**

You can use this feature to place compiled code and data at specific addresses in the RTX address space.

Code quality issues

This section describes the stratgies that the VFX C compiler uses to generate code. You can use the information contained in this section to write code which will run better on the RTX 2000 using the VFX C compiler.

Access to local data

The way that the compiler generates code to access local data depends upon a number of factors. The RTX 2010 supports fast access to 32 16-bit quantities in the frame without the need for changing the data page register (as the user page register is automatically selected when accessing user data). However,

all local data that spill over the 32 16-bit words cause the compiler to use a general-purpose fetch which sets the data page register and uses the same scheme as global data access, which leads to sub-optimal use of resources. Therefore, you should try to keep the amount of local data to less than 64 bytes.

The compiler allocates local 8-bit values a complete word in the frame. This is because the user-fetch and user-store instructions only work with 16-bit quantities. If, however, an 8-bit variable allocated in the frame has its address taken, the compiler does not generate user-fetch and user-store instructions to access the variable. Instead, it uses the general scheme of setting up the data page register before loading the variable indirectly.

Calling conventions

This section deals with the calling conventions used by the C compiler for the RTX 2010.

Current limitations

The current compiler does not support variadic functions, that is functions declared with a variable number of arguments using an ellipsis.

Caller/callee linkage

For prototyped functions, it is the callee's responsibility to remove all parameters from the data stack. This is natural for the RTX 2010 and leads to efficient function calling sequences.

It is the callee's ressponsibility to leave the UBR of the caller unchanged on function return — that is, the caller does not preserve its own UBR across a call. If a function must modify the UBR during execution (for example, to allocate locals), it is that function's responsibility to save and restore the UBR in is function prologue and epilogue.

Data page register

In small model the data page register retains its value throught the execution of a program. No code generated by the C compiler will change the value of DPR; if an assembler routine changes DRP, it must restore its value prior to the call for compiled code to work correctly.

In large model the data page register is always set before it is needed. Therefore, it is not necessary for any function to preserve the value of DPR during a function.

Parameter order

Parameters are passed in reverse order on the data stack, with the lexically-first argument pushed last and appearing at the caller as the topmost item on the data stack.

Simple parameters

In large and small model, all 8-bit and 16-bit quantities are passed on the data stack as 16-bit items.

Pointer parameters

In small model pointers are passed as 16-bit quantities representing the address within the data page. The data page is assumed because it does not change throughout the execution of the program.

In large model pointers are passed as 32-bit quantities. Topmost on the data stack is the page number which can be written directly to the data page register. Next on the stack is the 16-bit address within the data page.

Long parameters

All 32-bit quantities are passed with the topmost item on the stack as the most significant half of the 32-bit quantity and the next on stack as the least-significant half.

Floating-point parameters

Floating-point is not supported in the current software.

Structured parameters

Structured parameters are passed by reference with the called routine responsible for making a local copy of the input. The address of the data item is passed as pointer parameters for the current memory model.

Simple function return

All 8-bit and 16-bit data are returned on the data stack as a single item.

Pointer function return

In small model pointers are returned as 16-bit quantities representing the address within the data page. The data page is assumed because it does not change throughout the execution of the program.

In large model pointers are returned as 32-bit quantities. Topmost on the data stack is the page number, and next on the stack is the 16-bit address within the data page.

Long function return

All 32-bit quantities are returned with the topmost item on the stack as the most significant half of the 32-bit quantity and the next on stack as the least-significant half.

Floating-point return

Floating-point is not supported in the current software.

Structured returns

Structured parameters are returned by reference with the called routine responsible for making a copy of the returned value through an invisible pointer passed as the first argument to a function.Example

- **Example**

The compiler rewrites function calls with structured returns. Consider

```
/* Prototype */
SomeStruct foo(int x);

/* Structure assign of function result */
SomeStruct s = foo(1);
```

This is rewritten as:

```
/* Rewritten prototype with hidden parameter */
void foo(SomeStruct *hidden, int x);

/* Structure assign of function result */
SomeStruct s;
foo(&s, 1);
```

3

Compiling Files

Introduction

In contrast to many compilation and assembly language development systems, with VFX you don't invoke the assembler or compiler directly. Instead you'll normally use the *compiler driver* `hcl` as it provides an easy way to get files compiled, assembled, and linked. This section will introduce you to using the compiler driver to convert your source files to object files, executables, or other formats.

We recommend that you use the compiler driver rather than use the assembler or compiler directly because there the driver can assemble multiple files using one command line and can invoke the linker for you too. There is no reason why you should not invoke the assembler directly yourself, but you'll find that typing in all the required options is quite tedious — and why do that when `hcl` will provide them for you automatically?

Actually, the term compiler driver is a little misleading as `hcl` is capable of running the compilers for all VFX languages, the assembler, and the linker. You can turn to the compiler driver reference section to find about out the complete set of capabilities `hcl` offers.

Naming conventions

The compiler driver is capable of invoking compilers for a number of languages as well as running the assembler. We'll concentrate on C and assembler in this section, but you'll find a full feature list of the compiler driver in the reference section.

The compiler driver uses file extensions to distinguish the language the source file is written in. The compiler driver recognises the extension '`.c`' as C source files, '`.s`' and '`.asm`' as assembly code files, and '`.hzo`' as object code files.

We strongly recommend that you adopt these extensions for your source files and object files because you'll find that using the VFX tools is much easier if you do — as you'll see later.

C language files

When the compiler driver finds a file with a `.c` extension, it runs the C compiler to convert it to object code.

Assembly language files

When the compiler driver finds a file with a `.s` or `.asm` extension, it runs the assembler to convert it to object code.

Object code files

When the compiler driver finds a file with a `.hzo` extension, it passes it to the linker to include it in the final application.

Translating files

Translating a single file

The first thing that you'll probably want to do is assemble a single source file to get an object file. Suppose that you need to assemble the file `test.s` to an object file — you can do this using `hcl` as we suggested before, and all you type is:

```
hcl -c test.asm
```

The compiler driver invokes the assembler with all the necessary options to make it produce the object file `test.hzo`. The option `-c` tells `hcl` to assemble `test.s` to an object file, but to stop there and do nothing else.

To see what actually happens behind the scenes, you can ask `hcl` to show you the command lines that it executes by adding the `-v` option:

```
hcl -v -c test.asm
```

On my computer, the commands echoed to the screen are:

```
F:\VFX\Examples>hcl -v -c test.asm
F:\VFX\bin\has -JF:\VFX\include -I. test.s -o test.hzo
```

Now you can see why using the compiler driver is so much easier!

It doesn't matter about the order of the `-v` and `-c` switches, nor does it matter where you place them on the command line.

Assembling multiple files

Where the compiler driver really shines is that it can assemble more than one file with a single command line. Extending the example above, suppose you wish to assemble the three files `test.s`, `apdu.s`, and `handler.s` to three object files; this is no problem:

```
hcl -c test.s apdu.s handler.s
```

The compiler driver invokes the assembler three times to process the three source files. This sleight of hand is revealed using `-v`:

```
F:\VFX\Examples>hcl -v -c test.s apdu.s mem.s
F:\VFX\bin\has -JF:\VFX\include -I. test.s -o test.hzo
F:\VFX\bin\has -JF:\VFX\include -I. apdu.s -o apdu.hzo
F:\VFX\bin\has -JF:\VFX\include -I. mem.s -o mem.hzo
```

A rough guide to assembler options

If you wish to control the assembler directly, you will need to know about the options the assembler accepts. For those of you curious about such things we present them here.

Option	Description
`-D`*name=value*	Define assembler symbol *name* to be *value*
`-g`	Generate assembly-level debugging information
`-I`*path*	Add *path* to the list of paths to search for user include files
`-J`*path*	Add *path* to the list of paths to search for standard include files
`-o` *file*	Write object code to *file*
`-U`*name*	Undefine assembler symbol *name*
`-V`	Display tool version information
`-w`	Suppress warnings

Table 3-1 **Assembler option summary**

4

Source Structure

Introduction

In this section you'll be shown how to format the statements in assembly language files.

Writing assembly language statements

Assembly language files are constructed from assembly-language mnemonics and directives, collectively known as *source statements*. An assembly source module is a sequence of such source statements.

A statement is a combination of mnemonics, operands, and comments the defines the object code to be created at assembly time. Each line of source code contains a single statement.

Assembler statements take the form:

- **Syntax**

  [*label*] [*operation*] [*operands*] [*comment*]

Field	Purpose
name	Labels the statement so that the statement can be accessed by name in other statements
operation	Defines the action of the statement
operands	Defines the data to be operated on by the statement

Table 4-1 Fields used in assembler statements

Field	Purpose
comment	Describes the statement without having any affect on assembly

Table 4-1 Fields used in assembler statements

All fields are optional, although the operand or label fields may be required if certain directives or instructions are used in the operation field.

Label field

The label field starts at the left of the line, with no preceding spaces. A label name is a sequence of alphanumeric characters, starting with a letter. You can also use the dollar sign '$' and underline character '_' in label names.

A colon may be placed directly after the label, or it can be omitted. If a colon is placed after a label, it defines that label to be the value of the location counter in the current section.

Operation field

The operation field contains either a machine instruction or an assembler directive. You must write these either in all upper-case or all lower-case — mixed case is not allowed.

The operation field must not start at the left of the line; at least one space must precede it if there is no label field. At least one space must separate the label field and the operation field.

Operand field

The contents of the operand depend upon the instruction or directive in the operation field. Different instructions and directives have different operand field formats. Please refer to the specific section for details of the operand field.

Comment field

The comment field is optional, and contains information that is not essential to the assembler, but is useful for documentation. The comment field must be separated from the previous fields by at least one space.

Constants

You can use constants to specify numbers or strings that are set at assembly time.

Integer constants

Integer constants represent integer values and can be represented in binary, octal, decimal, or hexadecimal. You can specify the radix for the integer constant by adding a radix specified as a suffix to the number. If no radix specifier is given the constant is decimal.

■ **Syntax**

decimal-digt digit... [B | O | Q | D | H]

Radix	Specifier
Binary	B
Octal	O or Q
Decimal	D
Hexadecimal	H

Table 4-2 Integer constant radix suffixes

Radix suffixes can be given either in lower case or purchase letters.

Hexadecimal constants must always start with a decimal digit (0 to 9). You must do this otherwise the assembler will mistake the constant for a symbol — for example, `0FCH` is interpreted as a hexadecimal constant but `FCH` is interpreted as a symbol.

You can specify hexadecimal constants in an two other formats which are popular with many assemblers:

■ **Syntax**

`0x` *digit digit*...
`$` *digit digit*...

The `0x` notation is exactly the same as the way hexadecimal constants are written in C, and the `$` notation is common in many assemblers for Motorola parts.

■ **Examples**

```
224         ; 224
224D        ; 224
17Q         ; 15
100H        ; 256
0xC0        ; 192
$F          ; 15
```

String constants

A string constant consists of one or more ASCII characters enclosed in single or double quotation marks.

■ **Syntax**

'character...'
"character..."

You can specify non-printable characters in string constants using escape sequences. An escape sequence is introduced by \ (backslash). The following escape sequences are supported:

Escape sequence	Description
ooo	Octal code of character where *o* is an octal digit
\"	Double quotation mark
\\	Backslash
\'	Single quotation mark
\b	Backspace, ASCII code 8
\f	Form feed, ASCII code 12
\n	New line (line feed), ASCII code 10
\r	Carriage return, ASCII code 13
\t	Tab, ASCII code 9
\v	Vertical tab, ASCII code 11
\x*hh*	Hexadecimal code of character where *h* is a hexadecimal digit

Table 4-3 **Escape sequences in strings**

■ **Examples**

```
"This is a string constant"
'A string constant with a new line at the end\n'
```

Comments

To help others better understand some particularly tricky piece of code, you can insert comments into the source program. Comments are simply informational attachments and have no significance for the assembler.

Comments come in two forms: multi-line comments and single-line comments.

Single-line comments

A single line comment is introduced either by single character ; or by the characters //.

Syntax

```
// character...
; character...
```

The assembler ignores all characters from the comment introducer to the end of the line. This type of comment is particularly good when you want to comment a single assembler line.

Example

```
LOADI   dataPtr, 1          // fetch next byte from APDU
INCN    dataPtr             ;  increment data pointer
```

Which style you choose to use depends on your personal preference.

Multi-line comments

A multi-line comment resembles a standard C comment as it is introduced by the characters /* and is terminated by */.

Syntax

```
/* character... */
```

Anything in between these delimiters is ignored by the assembler. You can use this type of comment to place large amounts of commentary, such as copyright notices or functional descriptions, into your code.

Example

```
/* Elliptic curve cryptography library
   Copyright (c) 1999 Rowley Associates Limited.
*/
```

It's quite acceptable to use this form to comment a single line, but using the single-line comment form is stylistically better.

5

Labels and Variables

Introduction

This section explains how to define labels, variables, and other symbols that refer to data locations in sections. The assembler keeps an independent *location counter* for each section in your application. When you define data or code in a section, the location counter for that section is adjusted, and this adjustment does not affect the location counters of other sections. When you define a label in a section, the current value of the location counter for that section is assigned to the symbol.

You'll be shown how to assign labels and define most types of variables. You'll also be introduced to directives that control the location counter directly.

Label names

A label name is a sequence of alphanumeric characters, starting with a letter. You can also use the dollar sign '`$`' and underline character '_' in label names.

Symbolic constants or equates

You can define a symbolic name for a constant using the `EQU` directive. The `EQU` directive declares a symbol whose value is defined by an expression.

- **Syntax**

  symbol `EQU` *expression*
  symbol = *expression*

The assembler evaluates the expression and assigns that value to the symbol.

- **Example**

```
CR  EQU  13
```

This defines the symbol CR to be a symbolic name for the value 13. Now you can use the symbol CR in expressions rather than the constant.

The expression need not be constant or even known at assembly time; it can be any value and may include complex operations involving external symbols.

- **Example**

```
ADDRHI  EQU (ADDR>>8) & 0FFH
```

This defines the symbol ADDRHI to be equivalent to the value of ADDR shifted right 8 bits and then logically anded with 255. If ADDR is an external symbol defined in another module, then the expression involving ADDR cannot be resolved at assembly time as the value of ADDR isn't known to the assembler. The value of ADDR is only known when linking and the linker will resolve any expression the assembler can't.

Using type specifiers

Some statements need data type specifiers to define the size and type of an operand. Data type specifiers are fully described in Chapter 6, "Data Types" on page 35, but for the moment the following will suffice:

Type name	Size in bytes	Description
BYTE	1	Unsigned 8-bit byte
WORD	2	Unsigned 16-bit word
LONG	4	Unsigned 32-bit word
CHAR	1	8-bit character
ADDR	2	16-bit address

Table 5-1 Assembler built-in types

Labels

You use labels to give symbolic names to addresses of instructions or data. The most common form of label is a *code label* where code labels as operands of call, branch, and jump instructions to transfer program control to a new instruction. Another common form of label is a *data label* which labels a data storage area.

■ **Syntax**

label [:] [*directive* | *instruction*]

The label field starts at the left of the line, with no preceding spaces. The colon after the label is optional, but if present the assembler immediately defines the label as a code or data label. Some directives, such as EQU, require that you do not place a colon after the label.#

■ **Example**

```
ExitPt:   RET
```

This defines ExitPt as a code label which labels the RET instruction. You can branch to the RET instruction by using the label ExitPt in an instruction:

```
JMP ExitPt
```

Defining and initialising data

You can allocate and initialise memory for data using data definition directives. There are a wide range of data definition directives covering a wide range of uses, and many of these overlap.

Defining simple data

You can initialise data items which have simple types using a convenient set of directives.

■ **Syntax**

[DB | DW | DL | DD] *initializer* [, *initializer*]...

The size and type of the variable is determined by the directive. The directives used to define an object in this way are:

Directive	Meaning
DB	Define bytes
DW	Define words (2 bytes)
DD DL	Define long or double word (4 bytes)

Table 5-2 Simple data allocation directives

The label is assigned the location counter of the current section before the data are placed in that section. The label's data type is set to be an array of the given type, the bounds of which are set by the number of elements defined.

■ **Example**

```
Power10   DW    1, 10, 100, 1000, 10000
```

This defines the label `POWER10` and allocates five initialised words with the given values. The type of `POWER10` is set to `WORD[5]`, an array of five words, as five values are listed.

Defining string data

You can define string data using the `DB` directive. When the assembler sees a string, it expands the string into a series of bytes and places those into the current section.

■ **Example**

```
BufOvfl   DB  13, 10, "WARNING: buffer overflow", 0
```

This emits the bytes 13 and 10 into the current section, followed by the ASCII bytes comprising the string, and finally a trailing zero byte.

Aligning data

Data alignment is critical in many instances. Defining exactly how your data are arranged in memory is usually a requirement of interfacing with the outside world using devices or communication areas shared between the application and operating system.

You can align the current section's location counter with the `ALIGN` directive. This directive takes a single operand which must be a type name.

■ **Syntax**

`ALIGN` *type*

The data type given after the directive defines the alignment requirement; if this type has a size *n*, the location counter is adjusted to be divisible by *n* with no remainder.

■ **Example**

```
ALIGN  LONG
```

This aligns the location counter so that it lies on a 4-byte boundary as the type `LONG` has size 4.

To align data to an *n*-byte boundary you can use the following:

```
ALIGN BYTE[n]
```

Filling areas

In many cases you'll need to fill large areas of code or data areas with zeroes or some other value. The assembler provides the `.SPACE` and `.FILL` directives for this purpose.

Filling with zeroes

A particular example of this is reserving memory space for a large array. Rather than using multiple data definition directives, you can use the `.SPACE` directive.

- **Syntax**

  [*label*] `.SPACE` *n*

The `.SPACE` directive generates *n* bytes of zeroes into the current section and adjusts the location counter accordingly.

- **Example**

```
.SPACE  10
```

This reserves 10 bytes in the current section and sets them all to zero.

Filling with a particular value

You may find it convenient to use the `.FILL` directive to fill an area with a certain number of predefined bytes. `.FILL` takes two parameters, the number of bytes to generate and the byte to fill with.

- **Syntax**

  [*label*] `.FILL` *size*, *value*

The `.FILL` directive generates *size* bytes of *value* into the current section and adjusts the location counter accordingly.

- **Example**

```
.FILL  5, ' '
```

This generates five spaces into the current section.

6

Data Types

Introduction

Unlike many assemblers, the VFX assembler fully understands data types. The most well-known and widely-used assembler which uses data typing extensively is Microsoft's MASM — and its many clones. So, if you've used MASM before you should be pretty well at home with the concept of data types in an assembler and also with the VFX implementation of data typing.

If you haven't used MASM before you may well wonder why data typing should ever be put into an assembler, given that many assembly programs are written without the help of data types at all. But there *are* many good reasons to do so, even without the precedent set my Microsoft, and the two most valuable benefits are:

- The ability to catch potential or real errors at assembly time rather than letting them through the assembler to go undetected until applications are deployed in the field.
- Data typing is an additional and effective source of program documentation, describing the way that data are grouped and represented.

We don't expect you to fully appreciate the usefulness of assembly-level data typing until you've used it in an application and had first-hand experience of both benefits above. Of course, it's still possible to write (almost) typeless assembly code using the VFX assembler if you should wish to do so, but effective use of data typing is a real programmer aid when writing code.

Lastly, we should just mention one other important benefit that data typing brings and that is the interaction between properly-typed assembly code and the simulator. If you correctly type your data, the simulator will present the values held in memory using a format based on the type of the object rather

than as a string of hexadecimal bytes. Having source-level debugging information displayed in a human-readable format is surely a way to improve productivity.

Built-in types

The VFX assembler provides a number of built-in or predefined data types. These data types correspond to those you'd find in a high-level language such as C.

Type name	Size in bytes	Description
BYTE	1	Unsigned 8-bit byte
WORD	2	Unsigned 16-bit word
LONG	4	Unsigned 32-bit word
CHAR	1	8-bit character
ADDR	2	16-bit address

Table 6-1 **Assembler built-in types**

You can use these to allocate data storage; for instance the following allocates one word of data for the symbol `count`:

```
count   VAR   BYTE
```

The directive `DF` allocates one byte of space for `count` in the current section and sets `count`'s type to `BYTE`.

Structure types

Using the `STRUC` and `FIELD` directives you can define data items which are grouped together. Such a group is called a *structure* and can be thought of as a structure in C.

Structured types are bracketed between `STRUC` and `ENDSTRUC` and should contain only `FIELD` directives.

■ Example

From an unshameful British stance, we could declare a structure type called `Amount` which has two members, `Pounds` and `Pence` like this:

```
Amount   STRUC
Pounds   FIELD   LONG
Pence    FIELD   BYTE
         ENDSTRUC
```

The field Pounds is declared to be of type LONG and Pennies is of type BYTE (we can count lots of Pounds, and a small amount of loose change).

Structured allocation and field access

The most useful thing about structures, though, is that they act like any built-in data type, so you can allocate space for variables of structure type:

```
Balance   VAR    Amount
```

Here we've declared enough storage for the variable Balance to hold an Amount, and the assembler also knows that Balance is of type Amount. Because the assembler knows what type Balance is and how big an Amount is, you can load the contents of Balance onto the stack in a single instruction:

```
LOAD  Balance
```

This loads three bytes onto the stack (an amount is one word plus one byte).

What's more, because the assembler tracks type information, you can specify which members of balance to operate on:

```
LOAD  Balance.Pence
```

Here, we load a single byte onto the stack which is the Pence part of Balance. We could equally well have written:

```
LOAD  Balance.Pounds
```

which loads the Pounds part of Balance.

Nested structures

Because user-defined structures are no different from the built-in types, you can declare fields within other structures to be of structure type. Taking the example above a little further, you could define a type Account to have two members, Balance and ODLimit which correspond to an current account's balance and overdraft limit:

```
Account  STRUC
Balance  FIELD   Amount
ODLimit  FIELD   Amount
         ENDSTRUC

MyAccount DF     Account
```

Having a variable MyAccount declared of type Account, you can access the Pounds field of MyAccount's ODLimit member using the dotted field notation:

```
LOAD  MyAccount.ODLimit.Pounds
```

Array types

You can declare arrays of any predefined or user-defined type. Arrays are used extensively in high-level languages, and therefore we decided they should be available in the VFX assembler to make integration with C easier.

An array type is constructed by specifying the number of array elements in brackets after the data type.

- **Syntax**

  type [*array-size*]

This declares an array of *array-size* elements each of data type *type*. The *array-size* must be an absolute constant known at assembly time.

- **Example**

The type

```
BYTE[8]
```

declares an array of eight bytes.

Combining data types

Arrays, combined with structures, can make complex data structuring simple:

```
BankAccount  STRUC
HolderName   FIELD     CHAR[32]
HolderAddr   FIELD     CHAR[32]
Balance      FIELD     Amount
ODLimit      FIELD     Amount
             ENDSTRUC
```

Accessing individual array elements

You can select individual elements from an array by specifying the index to be used in brackets:

```
LOAD  MyAccount.HolderName[0]
```

The VFX assembler defines arrays as zero-based, so the fragment above loads the first character of `MyAccount`'s `HolderName`. Because the assembler must know the address to load at assembly-time, the expression in the brackets must evaluate to a constant at assembly time. For example, the following is invalid because `Index` isn't an assembly-time constant:

```
Index   DF    BYTE
        LOAD  MyAccount.HolderName[Index]
```

However, if `Index` were defined symbolically, the assembler can compute the correct address to encode in the instruction at assembly time:

```
Index   EQU   20
        LOAD  MyAccount.HolderName[Index]
```

Pointer types

You can declare pointers to types just like you can in most high-level languages.

Syntax

type `PTR`

This declares a pointer to the data type *type*.

Example

The type

```
BYTE PTR
```

declares a pointer to a byte. The built-in type `ADDR` is identical to the type `BYTE PTR`.

7

Expressions

Introduction

In this section you'll be introduced to the way in which you can write expressions in the assembler.

A note on operators and operands

Many conventional assemblers require that the values given to operators should be constant expressions known at assembly time. The VFX assembler does not require this to be so — if an expression cannot be evaluated by the assembler, the expression is passed to the linker for computation.

Byte and word extraction operators

These operators allow you to extract bytes from values at assembly time.

■ Syntax

The following table shows the syntax of the extraction operators and their meanings:

Operator	Syntax	Description
`HIGH` `HBYTE`	`HIGH` *expression* `HBYTE` *expression*	Extract the high byte of the 16-bit value *expression*
`LOW` `LBYTE`	`LOW` *expression* `LBYTE` *expression*	Extract the low byte of *expression*

Table 7-1 **Extraction operators**

Operator	Syntax	Description
HWORD	HWORD *expression*	Extract the high 16 bits of *expression*
LWORD	LWORD *expression*	Extract the low 16 bits of *expression*

Table 7-1 **Extraction operators**

■ **Examples**

```
HIGH  $FEDCBA98        ; evaluates to $BA
LOW   $FEDCBA98        ; evaluates to $98
HWORD $FEDCBA98        ; evaluates to $FEDC
LWORD $FEDCBA98        ; evaluates to $BA98
```

These can be combined to extract other bytes of values:

```
HBYTE HWORD $FEDCBA98        ; evaluates to $FE
LBYTE HWORD $FEDCBA98        ; evaluates to $DC
```

Index operator

The index operator indicates addition with a scale factor. It's similar to the addition operator.

■ **Syntax**

$expression_1$[$expression_2$]

$expression_1$ can be any expression which has array type. $expression_2$ must be a constant expression. The assembler multiplies $expression_2$ by the size of the array element type and adds it to $expression_1$.

■ **Example**

```
ARR   VAR   WORD[4]  ; an array of four words
W3    EQU   ARR[3]   ; set W4 to the address ARR+3*(SIZE WORD)
                      ; which is ARR+6
```

Bitwise operators

Bitwise operators perform logical operations on each bit of an expression.

■ **Syntax**

The following table shows the syntax of the logical operators and their meanings.

Don't confuse these operators with processor instructions having the same names. These operators are used on expressions at assembly time or link time, not at run time.

Operator	Syntax	Description
NOT ~	NOT *expression* ~ *expression*	Bitwise complement
AND &	$expression_1$ AND $expression_2$ $expression_1$ & $expression_2$	Bitwise and
OR |	$expression_1$ OR $expression_2$ $expression_1$ | $expression_2$	Bitwise inclusive or
XOR ^	$expression_1$ XOR $expression_2$ $expression_1$ ^ $expression_2$	Bitwise exclusive or

Table 7-2 Logical operators

Examples

```
NOT 0FH          ; evaluates to FFFFFFF0
0AAH AND 0F0H    ; evaluates to A0
0AAH OR 0F0H     ; evaluates to FA
0AAH XOR 0FFH     ; evaluates to 55
```

Arithmetic operators

Relational operators compare two expressions and return a true value if the condition specified by the operator is satisfied. The relational operators use the value one (1) to indicate that the condition is true and zero to indicate that it is false.

Syntax

The following table shows the syntax of the arithmetic operators and their meanings.

Operator	Syntax	Description
+	$expression_1$ + $expression_2$	True if expressions are equal
-	$expression_1$ - $expression_2$	True if expressions are not equal
*	$expression_1$ * $expression_2$	True if $expression_1$ is less than $expression_2$
/	$expression_1$ / $expression_2$	True if $expression_1$ is less than or equal to $expression_2$
SHL <<	$expression_1$ SHL $expression_2$ $expression_1$ << $expression_2$	True if $expression_1$ is greater than $expression_2$

Table 7-3 Arithmetic operators

Operator	Syntax	Description
SHR >>	*expression*$_1$ SHR *expression*$_2$ *expression*$_1$ >> *expression*$_2$	True if *expression*$_1$ is greater than or equal to *expression*$_2$

Table 7-3 **Arithmetic operators**

The right shift is an arithmetic shift.

■ Examples

```
1 EQ 2      ; evaluates to false (0)
1 NE 2      ; evaluates to true  (1)
1 LT 2      ; evaluates to true
1 GT 2      ; evaluates to false
1 LE 2      ; evaluates to true
1 GE 2      ; evaluates to false
```

Relational operators

Relational operators compare two expressions and return a true value if the condition specified by the operator is satisfied. The relational operators use the value one (1) to indicate that the condition is true and zero to indicate that it is false.

■ Syntax

The following table shows the syntax of the relational operators and their meanings.

Operator	Syntax	Description
EQ ==	*expression*$_1$ EQ *expression*$_2$ *expression*$_1$ == *expression*$_2$	True if expressions are equal
NE !=	*expression*$_1$ NE *expression*$_2$ *expression*$_1$!= *expression*$_2$	True if expressions are not equal
LT <	*expression*$_1$ LT *expression*$_2$ *expression*$_1$ < *expression*$_2$	True if *expression*$_1$ is less than *expression*$_2$
LE <=	*expression*$_1$ LE *expression*$_2$ *expression*$_1$ <= *expression*$_2$	True if *expression*$_1$ is less than or equal to *expression*$_2$
GT >	*expression*$_1$ GT *expression*$_2$ *expression*$_1$ > *expression*$_2$	True if *expression*$_1$ is greater than *expression*$_2$

Table 7-4 **Relational operators**

Operator	Syntax	Description
GE >=	*expression*$_1$ GE *expression*$_2$ *expression*$_1$ >= *expression*$_2$	True if *expression*$_1$ is greater than or equal to *expression*$_2$

Table 7-4 **Relational operators**

Relational operators treat their operands as signed 32-bit numbers, so `0FFFFFFFFH GT 1` is false but `0FFFF GT 1` is true.

■ Examples

```
1 EQ 2      ; evaluates to false (0)
1 NE 2      ; evaluates to true  (1)
1 LT 2      ; evaluates to true
1 GT 2      ; evaluates to false
1 LE 2      ; evaluates to true
1 GE 2      ; evaluates to false
```

Logical operators

Logical (Boolean) operators operate on expressions to deliver logical results.

■ Syntax

The following table shows the syntax of the logical operators and their meanings.

Operator	Syntax	Description
AND &&	*expression*$_1$ AND *expression*$_2$ *expression*$_1$ && *expression*$_2$	True if both *expression*$_1$ and *expression*$_2$ are true
OR ||	*expression*$_1$ OR *expression*$_2$ *expression*$_1$ || *expression*$_2$	True if either *expression*$_1$ or *expression*$_2$ are true
LNOT !	LNOT *expression* ! *expression*	True if *expression* is false, and false if *expression* is true

Table 7-5 **Logical operators**

■ Examples

```
1 AND 0      ; evaluates to false (0)
1 AND 1      ; evaluates to true  (1)
1 OR 2      ; evaluates to true
NOT 1       ; evaluates to false
```

Miscellaneous operators

Retype operator

The retype operator `::` allows you to override the data type of an operand, providing the operand with a new type.

- **Syntax**

  expression `::` *type*

The expression is evaluated and given the type, replacing whatever type (if any) the expression had.

- **Example**

```
wordvar DW 2

        LOAD  wordvar::BYTE
```

In this example, `wordvar` has the type `WORD` because it is defined using `DW`. The load, however, loads only a single byte because `wordvar` is retyped as a `BYTE`. Because retyping does not alter the value of the expression, it only alters its type, the load will read from the lowest address of `wordvar`.

This operator

You can refer to the current value of the location counter without using a label using the `THIS` operator.

- **Syntax**

```
THIS
```

The `THIS` operator returns an expression which denotes the location counter *at the start of the assembler line*. This is very important: the location counter returned by `THIS` does not change even if code is emitted.

- **Example**

A typical use of `THIS` is to compute the size of a string or block of memory.

```
MyString    DB   "Why would you count the number of characters"
            DB   "in a string when the assembler can do it?"

MyStringLen EQU   THIS-MyString
```

Defined operator

You can use the `DEFINED` operator to see whether a symbol is defined or not. Typically, this is used with conditional directives to control whether a portion of a file is assembled or not.

Syntax

`DEFINED` *symbol*

The `DEFINED` operator returns a Boolean result which is true of the symbol is define *at that point in the file*, and false otherwise. Note that this operator only

8

Sections

Introduction

You can use sections to logically separate pieces of code or data. For instance, many processors need interrupt and reset vectors placed at specific addresses; using sections you can place data at these addresses. Another use of sections is to separate volatile data from non-volatile data in an embedded system with dynamic RAM and battery-backed static RAM.

Selecting a section

You select a section using the `.SECT` directive. If the section name doesn't exist, a new section is created.

Syntax

`.SECT "`*section-name*`"`

By convention, section names start with a period character. The section name is not an assembler symbol and can't be used in any expression.

The assembler places all code and data into the selected section until another section is selected using `.SECT`.

Example

```
        .SECT  ".vectors"
irqvec  DW     irq
nmivec  DW     nmi
rstvec  DW     reset
```

This example shows you how to set up a section named `.vectors` and populate it with some address.

Predefined sections

The VFX system uses a number of predefined sections:

Section name	Description
`.text`	Code section
`.data`	Initialised data section
`.bss`	Uninitialised data section

Table 8-1 Predefined sections

You can, however, instruct the compiler to place code on a per-function basis and data on a per-object basis into a user-defined section. You can use this feature to fix code and data at specific addresses to construct jump tables, vector tables, and place data into non-volatile memory.

9

Conditional Assembly

Introduction

Conditional assembly allows you to control which code gets assembled as part of your application.

General structure

The structure of conditional assembly is much like that used by high-level language conditional constructs and the C pre-processor. The syntax you use is:

- **Syntax**

```
IF expression
  true-conditional-body
ENDIF
```

or

```
IF expression
  true-conditional-body
ELSE
  false-conditional-body
ENDIF
```

The controlling expression must be an absolute assembly-time constant. When the expression is non-zero the true conditional arm is assembled; when the expression is zero the false conditional body, if any, is assembled.

Cascading conditionals

You'll find that using the `IF` and `ENDIF` directives on their own sometimes produces code which is difficult to follow.

Example

Consider the following which has nested `IF` directives:

```
IF type == 1
  CALL type1
ELSE
  IF type == 2
    CALL type2
  ELSE
    CALL type3
  ENDIF
ENDIF
```

The nested conditional can be replaced using the `ELIF` directive which acts like `ELSE IF`:

```
IF type == 1
  CALL type1
ELIF type == 2
  CALL type2
ELSE
  CALL type3
ENDIF
```

The full formal syntax for using `IF`, `ELIF`, and `ENDIF` is:

Syntax

`IF` *expression*
  statements
{ `ELIF`
  statements }
[`ELSE`
  statements]
`ENDIF`

Typical uses

There are a few typical ways which you can use the conditional assembly feature. Mostly these will involve turning on or off a piece of code according to some criterion, for example to assemble a piece of code with debugging enabled, or to assemble a piece of code for a specific variant of operating system.

Omitting debugging code

When you write an application you'll invariable debug it by either using a debugger or by inserting special debugging code. When the final application is deployed, you'll want the application to be delivered with the debugging code removed — it takes up valuable resources and won't be necessary in the final application. However, it's always nice to keep the debugging code around just in case you should need to test the application again.

■ Example

Usual practice is to use a symbol, `_DEBUG`, as a flag to either include or exclude debugging code. Now you can use `IF` with the `DEFINED` operator to conditionally assemble some parts of your application depending upon whether the `_DEBUG` symbol is defined or not. You can define `_DEBUG` on the compiler driver or assembler command line.

```
IF DEFINED _DEBUG
  CALL DumpAppState
ENDIF
```

You can control whether the call to `DumpAppState` is made by defining the symbol `_DEBUG`. You can do this by defining the symbol in the assembler source file, or more flexible would be to define the symbol when assembling the file by using the compiler driver's `-D` option.

```
hcl -c -D_DEBUG app.asm
```

To assemble the application for production you would leave the `_DEBUG` symbol undefined and assemble using:

```
hcl -c app.asm
```

Targeting specific environments

Another common use of conditional assembly is to include or exclude code according to the intended target operating system. For example, an application may need to be deployed on two versions of an operating system but should take advantage of some extended features in one of them.

You can do this by defining a symbol and setting its value on the command line. Then, inside the application, you can examine the value of this symbol to conditionally assemble a piece of code.

■ Example

A feature called *transaction protection* only exists in MULTOS version 4 and later; earlier versions of MULTOS do not support it. An application may need to be targeted to versions of MULTOS both with and without transaction

protection, for example during a transition from one MULTOS card to another. We can use the conditional assembly feature to target both MULTOS versions and take advantage of the transaction protection feature in version 4.

The following code uses the value of `__MULTOS_VERSION` to conditionally assemble the calls to `TransProtOn` and `TransProtOff` which turn transaction protection on and off for MULTOS 4 and later:

```
IF __MULTOS_VERSION >= 4
  CALL TransactionProtectionOn
ENDIF
INCN  TransNum
IF __MULTOS_VERSION >= 4
  CALL TransactionProtectionOff
ENDIF
```

You set the specific version of MULTOS to target by defining the `__MULTOS_VERSION` symbol on the command line. To assemble the code for MULTOS version 3 and therefore exclude transaction protection you would use:

```
hcl -c -D__MULTOS_VERSION=3 trans.asm
```

And to assemble for MULTOS version 4 and include transaction protection you would use:

```
hcl -c -D__MULTOS_VERSION=4 trans.asm
```

10

Using Multiple Modules

Introduction

When applications grow large they are usually broken into multiple smaller, manageable pieces (usually called *modules*). Each piece is assembled separately and then the pieces are stitched together by the linker to produce the final application.

When you partition a application into separate modules you will need to indicate how a symbol defined in one module is referenced in other modules. This section will show you how to declare symbols *exported* or *imported* so they can be used in more than one module.

Exporting symbols

Only symbols exported from a module can be used by other modules. You can export symbols using the `.EXPORT` directive. This directive does nothing more than make the symbol visible to other modules: it does not reserve storage for it nor define its type.

- **Syntax**

  `.EXPORT` *symbol* [`,` *symbol*]...

Not all symbols can be exported. Variables, labels, function blocks, and numeric constants defined using `EQU` can be exported, but macro names and local stack-based variables cannot.

The VFX assembler publishes the symbol in the object file so that other modules can access it. If you don't export a symbol you can only use it in the source file it's declared in.

A notational convenience

As a convenience, a label can be defined and exported at the same time using double-colon notation.

- **Example**

```
data_ptr::
```

This declares the label `data_ptr` and exports it. This is equivalent to:

```
      .EXPORT data_ptr
data_ptr:
```

Importing symbols

When you need to use symbols defined in other modules you must import them first. You import symbols using the `.IMPORT` directive.

- **Syntax**

`.IMPORT` *symbol* [`::` *type*] [`,` *symbol* [`::` *type*]]...

When importing a symbol you can also define its type. This type information is used by the assembler whenever you reference the imported symbol and acts just like a symbol declared locally within the module. If you don't define a type for the imported variable, no type information is available to the assembler. If you subsequently use such a variable where type information is required, the assembler will report an error.

- **Example**

```
.IMPORT  CLA::BYTE, La::WORD
.IMPORT  APDUData::BYTE[256]
.IMPORT  _myVar
```

The above imports `CLA` as a byte, `La` as a 16-bit word, `APDUData` as an array of 256 bytes, and `_myVar` without any type information.

Other terminology for import and export

The terms import and export are not the only ones used to define symbol linkage between modules. If you've programmed in other assembly languages or used other assemblers before you may be more familiar with the terminology *public symbols* and *external symbols*. You can use the following synonyms for `.IMPORT` and `.EXPORT`:

Directive	Synonyms
.IMPORT	.EXTERN, XREF
.EXPORT	.PUBLIC, XDEF

Table 10-1 Import/export directive synonyms

Libraries

When you know that you will need routines is a specific library you can use the INCLUDELIB directive to include them. Using INCLUDELIB means that you don't need to specify the library name on the compiler driver command line when you link your program.

- **Syntax**

```
INCLUDELIB "libname"
```

libname is the name of the library you'd like the linker to include at link time. The above syntax is equivalent to the linker switch "-l*libname*."

- **Example**

```
INCLUDELIB "java"
```

The above will cause the linker to search for and link the library libjava.hza.

The VFX compilers use this ability to transparently ask the linker to include the necessary runtime support package for the given language. The following libraries are automatically included when using object files produced by one of the VFX compilers.

Library	Language
libc.hza	C
libbasic.hza	BASIC
libjava.hza	Java
libforth.hza	Forth

Table 10-2 Language libraries linked automatically

11

RTX Assembler Reference

Introduction

This section describes the RTX assembler mnemonics used by the VFX tools. It is particularly useful if you need to write small RTX assembler routines to augment your application.

This document does not describe the RTX architecture in detail; you will need to refer to the *RTX 2000 Programmers Reference Manual* for further details.

Shifter instructions

This section contains a summary of the syntax of RTX shifter instructions.

_NOP

```
_NOP
```

No shifter operation.

_0LESS

```
_0LESS
```

Sign extend TOS.

_2STAR

```
_2STAR
```

Logical shift left TOS.

_2STARC

```
_2STARC
```

Rotate left TOS.

_CU2SLASH

```
_CU2SLASH
```

Logical shift right TOS out of carry.

_C2SLASH

```
_C2SLASH
```

Rotate TOS right through carry.

_U2SLASH

```
_U2SLASH
```

Logical shift right TOS.

_2SLASH

```
_2SLASH
```

Arithmetic shift right TOS.

_N2STAR

```
_N2STAR
```

Arithmetic shift left NOS.

_N2STARC

```
_N2STARC
```

Rottate left NOS.

_D2STAR

_D2STAR

Shift left TOS:NOS.

_D2STARC

_D2STARC

Rotate left TOS:NOS.

_CUD2SLASH

_CUD2SLASH

Shift right TOS:NOS right out of carry.

_CD2SLASH

_CUD2SLASH

Rotate right TOS:NOS through carry.

_UD2SLASH

_CUD2SLASH

Logical shift right TOS:NOS right.

_D2SLASH

_CUD2SLASH

Arithmetic shift right TOS:NOS right.

ALU operations

NOT

NOT

Invert TOS.

AND

```
AND
```

Bitwise-and NOS with TOS.

NOR

```
NOR
```

Bitwise-nor NOS with TOS.

XSUB

```
XSUB
```

Subtract NOS from TOS.

XSUBB

```
XSUBB
```

Subtract NOS from TOS with borrow.

OR

```
OR
```

Bitwise-or NOS with TOS.

NAND

```
NAND
```

Bitwise-nand NOS with TOS.

ADD

```
ADD
```

Add NOS to TOS.

ADDC

```
ADDC
```

Add NOS to TOS with carry.

XOR

```
ADD
```

Bitwise-xor NOS with TOS.

XNOR

```
XNOR
```

Bitwise-xnor NOS with TOS.

SUB

```
XSUB
```

Subtract TOS from NOS.

SUBB

```
XSUBB
```

Subtract TOS from NOS with borrow.

Instructions

This section contains a summary of the syntax of RTX instructions.

GSTORE

```
GSTORE          addr
```

Store top of stack into gbus address *addr*.

GFETCH

```
GFETCH          addr
```

Fetch gbus address *addr* to the top of stack.

USTORE

```
USTORE          addr
```

Store top of stack into user memory at *addr.*

UFETCH

```
UFETCH          addr
```

Fetch user memory at *addr* to the top of stack.

LIT

```
LIT             n
```

Push *n* to stack. *n* can be a relocatable expression.

CALL

```
CALL            addr
```

Call *addr.* The linker encodes this instruction using the optimal encoding sequence, either using a short call or a programmed long call as appropriate.

B

```
B               addr
```

Branch to *addr.* The linker encodes this instruction using the optimal encoding sequence, either using a short branch or a programmed long branch as appropriate.

BZ

```
BZ              addr
```

Branch to *addr* if TOS is zero, don't pop stack. The linker encodes this instruction using the optimal encoding sequence, either using a short branch or a programmed long branch as appropriate.

BZP

BZP *addr*

Branch to *addr* if TOS is zero, pop stack. The linker encodes this instruction using the optimal encoding sequence, either using a short branch or a programmed long branch as appropriate.

BINZ

BINZ *addr*

Branch to *addr* if INDEX is not zero. The linker encodes this instruction using the optimal encoding sequence, either using a short branch or a programmed long branch as appropriate.

UCODE

UCODE *insn*

Encode *insn* inline as an instruction. This is equivalent to DC2 *insn*, except that the lister will disassemble ucodes, whereas it will simply list the valu encoded for a DC2 directive.

SELDPR

SELDPR

All data access directed through the DPR.

SELCPR

SELCPR

All data access directed through the CPR.

SOFTINT

SOFTINT

Software interrupt.

CLRSOFTINT

```
CLRSOFTINT
```

Clear software interrupt status.

MULU

```
MULU
```

Single-cycle unsigned multiply. See the RTX 2010 data sheet for further information.

MULS

```
MULS
```

Single-cycle signed multiply. See the RTX 2010 data sheet for further information.

TOR

```
TOR
```

Equivalent to `GSTORE 1`. Move the top endtry on the data stack to the return stack.

ZEROEQ

```
ZEROEQ
```

Set TOS to -1 if TOS is zero, and to zero otherwise.

RFROM

```
RFROM
```

Equivalent to `GFETCH 1`. Move the top endtry on the data stack to the return stack.

NIP

```
NIP
```

Drop NOS from stack (SWAP DROP).

DUP

```
DUP
```

Duplicate TOS.

DROP

```
DROP
```

Drop TOS.

SWAP

```
SWAP
```

Swap order of TOS and NOS.

OVER

```
OVER
```

Duplicate NOS into TOS.

CFETCH

```
CFETCH
```

Fetch byte at TOS to TOS.

FETCH

```
FETCH
```

Fetch word at TOS to TOS.

CSTORE

```
CSTORE
```

Store byte in NOS at address in TOS.

STORE

```
STORE
```

Store word in NOS at address in TOS.

NEGATE

```
NEGATE
```

Equivalent to `LIT 0 XSUB`.

RETURN

```
RETURN
```

Return from subroutine call.

12

Compiler Driver Reference

Introduction

This section describes the switches accepted by the VFX compiler driver, `hcl`. The VFX compiler driver is capable of controlling compilation by all supported language compilers and the final link by the VFX linker.

Command syntax

You invoke the compiler driver using the following syntax:

`hcl` [*option* | *file*]...

Files

file is a source file to compile. The compiler driver uses the extension of the file to invoke the appropriate compiler. The compile driver supports the following file types, but note that not all compilers are available for all architectures:

Extension	Compiler invoked
`.fth`	Forth compiler, `hfc`.
`.c`	C compiler, `hcc`.
`.bas`	Basic compiler, `hbc`.
`.java`	Java compiler, `hjc`.
`.s .asm`	Assembler, `has`.

Table 12-1 Languages and file extensions

Extension	Compiler invoked
`.hzo`	None, object file is sent to linker, `hld`.

Table 12-1 Languages and file extensions

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated.

The compiler supports the following command line options:

Option	Description
`-A`	Warn about potential ANSI problems
`-c`	Compile to object code, do not link
`-D`*name* `-D`*name=val*	Define the preprocessor symbol *name* and optionally initialise it to *val*
`-E`*name*	Set program entry symbol to *name*
`-F`*fmt*	Set linker output format to *fmt*
`-g`	Generate symbolic debugging information
`-I`*dir*	Add *dir* to the end of the user include search list
`-J`*dir*	Add *dir* to the end of the system include search list
`-L`*dir*	Set the library search directory to *dir*
`-l`*x*	Search library *x* to resolve symbols
`-M`	Display linkage map on standard output
`-n`	Dry run — do not run compilers, assembler, or linker
`-N`	Do not search standard directories for include files
`-Ns`	Do not link the standard startup file `crt0.hzo`.
`-O`	Optimize output
`-o` *file*	Leave output in *file*
`-T`*name=l-addr*[*/r-addr*]	Set section *name* to load at *l-addr* and run at *r-addr*
`-U`*name*	Undefine preprocessor symbol *name*

Table 12-2 Compiler driver command line options

Option	Description
`-v`	Show commands as they are executed
`-V`	Display tool versions on execution
`-w`	Suppress warnings
`-W[cjfbal]`*arg*	Pass *arg* to compiler, assembler, or linker

Table 12-2 Compiler driver command line options

Verbose execution

The compiler driver and compilers usually operate without displaying any information messages or banners—only diagnostics such as errors and warnings are displayed.

If the `-V` switch is given, the compiler driver displays its version and it passes the switch on to each compiler and the linker so that they display their respective versions.

The `-v` switch displays the command lines which are executed by the compiler driver.

Where to find included files

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the `-I` switch which is passed on to each of the language processors.

Setting the output format

The linker supports a number of industry-standard file formats and also the native VFX file format. The compiler driver arranges for the linker to generate whatever is the most appropriate format for the tool set in use. For the MEL target, the default is to write the application as an unencrypted application load unit.

The `-F`*fmt* switch sets the output format, and the following formats are supported:

Switch	Format description
`-Fsrec`	Motorola S-records
`-Fhex`	Intel extended hex

Table 12-3 Supported output formats

Switch	Format description
-Fppx	Stag hex
-Ftek	Tektronix hex
-Fhzx	VFX native
-Falu	MULTOS ALU (MEL compiler only)

Table 12-3 Supported output formats

Macro definitions

Macros can be defined using the -D switch and undefined using the -U switch. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language.

The -D switch takes the form:

-D*name*

or

-D*name=value*

The first defines the macro *name* but without an associated replacement value, and the second defines the same macro with the replacement value *name*.

The -U switch is similar to -D but only supports the format

-U*name*

C compiler

When passed to the C compiler, the macros are interpreted by the compiler's pre-processor according to the ANSI standard. The pre-processor interprets a macro defined using -D*name* as equivalent to the declaration:

```
#define name
```

And and a macro defined using -D*name=value* is equivalent to:

```
#define name value
```

A macro undefined using -U*name* is equivalent to

```
#undef name
```

Forth compiler

When passed to the Forth compiler, -D*name* is equivalent to the declaration

```
TRUE CONSTANT name
```

And `-D`*name*`=`*value* is equivalent to the declaration:

```
value CONSTANT name
```

A macro undefined using `-U`*name* is equivalent to

```
FALSE CONSTANT name
```

Java compiler

The Java compiler accepts macro definition switches but ignores them.

Linking in libraries

You can link in libraries using the `-l`x option. You need to specify the directory where libraries are to be found, and you can do this with the `-L` option.

- **Example**

Link the library `libcrypto.hza` and `libio.hza` found in `/usr/VFX/lib`:

```
hcl -L/usr/VFX/lib -lcrypto -llibio
```

13

The Linker

Introduction

The linker `hld` is responsible for linking together the object files which make up your application together with some runtime startup code and any support libraries.

Although the compiler driver usually invokes the linker for you, we fully describe how the linker can be used stand-alone. If you're maintaining your project with a make-like program, you may wish to use this information to invoke the linker directly rather than using the compiler driver.

Linker function and features

The linker performs the following functions:

- resolves references between object modules;
- extracts object modules from archives to resolve unsatisfied references;
- combines all fragments belonging to the same section into a contiguous region;
- removes all unreferenced code and data;
- runs an architecture-specific optimizer to improve the object code;
- fixes the size of span-dependent instructions;
- computes all relocatable values;
- produces a linked application which can be in a number of formats.

The VFX tools were designed to be flexible and cover a wide range of processors, and also to let you to easily write space-efficient programs. To that end, the assembler and linker combination provides a number of features which are not found in many compilation systems.

Optimum-sized branches

The linker automatically resizes branches to labels where the label is too far away to be reached by a branch instruction. This is completely transparent to you as a programmer—when you use branch instructions, your linked program will always use the smallest possible branch instruction. This capability is deferred to the linker so that branches across compilation units are still optimized.

Fragments: killing dead code and data

The most important features of the linker are its ability to leave all unreferenced code and data out of the final application and to optimize the application as a whole, rather than on a per-function basis. The linker automatically discards all code and data fragments in a program which are not reachable from the start symbol.

Command syntax

You invoke the linker using the following syntax:

`hld` [*option* | *file*]...

- **Files**

file is an object file to include in the link and it must be in VFX object format. You do not give library files on the command line in this way, you specify then using the `-l` and `-L` switches described below.

- **Options**

option is a command-line option. Options are case sensitive and cannot be abbreviated.

The linker supports the following command line options:

Option	Description
`-E`*name*	Set program entry symbol to *name*
`-F`*fmt*	Set linker output format to *fmt*

Table 13-1 Object file lister option summary

Option	Description
-L*dir*	Set the library search directory to *dir*
-l*x*	Search library *x* to resolve symbols
-M	Display linkage map on standard output
-O	Optimize output
-o *file*	Leave output in *file*
-V	Display version

Table 13-1 Object file lister option summary

Linker output formats

The linker supports a number of industry-standard file formats and also the native VFX file format. These formats are selected using the -F*fmt* switch as follows:

Format switch	Format description
-Fmot	Motorola S-record
-Fhex	Intel extended hex
-Ftek	Tektronix hex
-Fppx	Stag hex
-Flst	Hex dump
-Fhzx	VFX native
-Falu	MULTOS ALU

Table 13-2 Supported output formats

Laying out memory

The linker need to know where to place all code and data which make up an application. You tell the linker where to place each section using the -T switch.

- **Example**

```
hld app.hzo -T.text=200 -T.bss=4000 -T.data=8000
```

This sets the .text section to start at address 200_{16}, the .bss section from 4000_{16}, and the .data section from 8000_{16}.

■ **Example**

You can assign where your own sections are placed. Assume you need to place the following set of vectors at address $FFFA_{16}$ as this is where the processor expects to find them.

```
.SECT ".vectors"
DW    irq
DW    nmi
DW    reset
```

You use the `-T` switch as above to set the address:

```
hld -T.vectors=fffa
```

Linkage maps

You can find where the linker has allocated your code and data in sections by asking for a linkage map using the `-M` option. The map is a listing of all public symbols with their addresses.

Linking in libraries

You can link in libraries using the `-l`x option. You need to specify the directory where libraries are to be found, and you can do this with the `-L` option.

■ **Example**

Link the library `libcrypto.hza` and `libio.hza` found in `/usr/VFX/lib`:

```
hld -L/usr/VFX/lib -lcrypto -llibio
```

14

The Archiver

Introduction

This section describes the archiver, or librarian, which you can use to create object code libraries. Object code libraries are collections of object files which are consolidated into a single file, called an archive. The benefit of an archive is that you can pass it to the linker and the linker will search the archive to resolve symbols needed during a link.

By convention, archives have the extension '`.hza`.' In fact, the format used for archives is compatible with PKWare's popular Zip format with deflate compression, so you can manipulate and browse VFX archives using many readily-available utilities for Windows.

Command syntax

You invoke the archiver using the following syntax:

`har` [*option*] *archive file*...

- **Files**

archive is the archive to operator on. *file* is an object file to add, replace, or delete from the archive according to *option*.

- **Options**

option is a command-line option. Options are case sensitive and cannot be The options which are recognised by the archiver are:

Switch	Description
-a	Add member to archive
-c	Create archive
-d	Delete member from archive
-r	Replace member in archive
-t	List members of archive
-v	Verbose mode
-V	Display version of archiver

Table 14-1 Archiver options

Archive operations

This section will take you through the typical operations you'll use the archiver for.

Creating an archive

To create an archive you simply use `har` with the name of the new archive and list any files which you wish it to contain. The archive will ne created, overwriting any archive which already exists with the same name.

- **Example**

To create an archive called `cclib.hza` which initially contains the two object code files `ir.hzo` and `cg.hzo` you would use:

```
har -c cclib.hza ir.hzo cg.hzo
```

The archiver expands wildcard file names so you can use

```
har -c cclin.hza *.hzo
```

to create an archive containing all the object files found in the current directory.

Listing the members of an archive

To show the members which comprise an archive, you use the `-v` switch. The member's names are listed together with their sizes. If you only give the archive name on the command line, the archiver lists all the members contained in the archive. However, you can list the attributes of specific members of the achive by specifying the names of the members you're interested in by specifying them on the command like

■ **Example**

To list all the members of the archive `cclib.hza` created above you'd use:

```
har -t cclib.hza
```

To list only the attributes of the member `ir.hzo` contained in the archive `cclib.hza` you'd use:

```
har -t cclib.hza ir.hzo
```

Adding and replacing members of an archive

You can replace members in an archive using the `-r` switch. If you specify a file to add to the archive, and the archive doesn't contain that member, the file is simply appended to the archive as a new member.

■ **Example**

To replace the member `ir.hzo` in the archive `cclib.hza` with the file `ir.hzo` on disk you would use:

```
har cclib.hza -r ir.hzo
```

Removing members from an archive

You can remove members from the archive using the `-d` switch, short for *delete*.

■ **Example**

To remove the member `ir.hzo` from the archive `cclib.hza` you'd use:

```
har cclib.hza -d ir.hzo
```

15

The Object File Lister

Introduction

The object file lister `hls` is a general-purpose program which can display useful information held in unlinked object files and linked executables. It can:

- show disassembled code
- show section addresses and sizes
- show exported symbols with their types and values
- show imported symbols with their types

Typically it's used to show object code generated by the VFX compilers, or to produce an intermixed listing of code and high-level source.

Command syntax

You invoke the object filer using the following syntax:

`hls` [*option*]... *file*

Files

file is the object file to list.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated.

The object file lister supports the following command line options

Option	Description
-a	Show all object bytes generated in the listing
-g	Generate an intermixed source listing
-p	Show public symbols defined in the module
-s	Show all symbols defined or used in the module
-S*path*	Add *path* to the list of paths to search for source files
-t	Display section start address, end address, and size
-x	Show external symbols referenced by the module
-V	Display tool version information

Table 15-1 Object file lister option summary

Listing section attributes

You can quickly see a summary of each section in an executable using the -t switch. This presents a summary for each section showing its start address, its end address, its size in decimal and hexadecimal, and the section name.

- **Example**

```
hls -t app.hzx
```

This lists the contents of all sections in the file app.hzx.

16

The Hex Extractor

Introduction

The hex extractor `hex` is used to prepare images in a number of formats to burn into EPROM or flash memory. Although the linker is capable of writing all the formats described here, it doesn't have the capability of splitting files for different bus or device sizes.

Command syntax

You invoke the hex extractor using the following syntax:

`hex` [*option*]... *file*

■ Files

file is the object file to convert.

■ Options

option is a command-line option. Options are case sensitive and cannot be abbreviated.

The hex extractor supports the following command line options

Option	Description
`-B`*n*	Split file at *n* kilobit boundaries
`-F`*fmt*	Select output format *fmt*

Table 16-1 Hex extractor option summary

Option	Description
-K*n*	Split file at *n* kilobyte boundaries
-o *file*	Write output with *file* as a prefix
-R*range*	Extract specified range from input file
-T*name*	Extract section *name* from input file
-V	Display tool version information
-W*width*	Set bus width to *width*

Table 16-1 Hex extractor option summary

Supported output formats

The following formats are supported:

- Straight binary
- Intel hex, also called Intellec format
- Morotola S-records, also called Exorcisor format
- Extended Tektronix Hex
- ASCII hex dump.

Preparing images for download

When you prepare to download applications to a monitor held in ROM, you usually need the application in a single industry-standard format such as S-records or Intellec hex format. What you don't need to do is split high and low bytes, and you won't need to split across ROMs.

The extractor generates files in this format by default — all you need to provide is the format you need the file in.

Example

```
hex -Fhex app.hzx
```

This will generate a single Intellect-format file, `app.hzx.hex`, which contains all code and data in the application. The addresses in the output file are the physical addresses of where the code and data are to be loaded.

Preparing images for ROM

Preparing ROM images is more involved than preparing images to download using a monitor. You may need to split images across multiple ROMs, for instance to separate high and low bytes for a 16-bit bus, or to split a large application across multiple ROMs because a single ROM doesn't have the capacity to hold the whole application.

The extractor copes with both of these requirements.

- **Example**

```
hex -B256 app.hzx -Fhex
```

This splits the application file `app.hzx` into multiple pieces, each of which is 256 kilobits. The `-B` option matches the part names of most 8-bit EPROMs, so the application is split into 256 kilobit sections, or 32 kilobyte chunks. `-B256` can equally-well have been specified using `-K32`.

- **Example**

```
hex -W2 app.hzx -Fhex
```

This splits the application file `app.hzx` into high and low bytes for a processor with a 16-bit bus. The bytes at even addresses are placed in one file, and the bytes at odd addresses in a second.

- **Example**

```
hex -W2 -B512 app.hzx -Fhex
```

This splits the application file `app.hzx` into high and low bytes for a processor with a 16-bit bus and also splits each of these into ROMs which have a 512 kilobit capacity. The bytes at even addresses are placed in one set of files, and the bytes at odd addresses in a second set of files.

Extracting parts of applications

By default the extractor will generate output files which contain the whole application. In some cases you may need to extract only the contents of certain sections or fixed range of addresses.

Extracting sections

If you need only to extract only certain sections from a file, you can use the `-T` option to specify those to extract.

■ **Example**

```
hex -T.text -T.vectors app.hzx
```

This will generate a single file, `app.hzx.bin`, which contains only the sections `.text` and `.vectors` from the file `app.hzx`.

Extracting ranges

If you need only to extract only certain ranges of addresses a file, you can use the `-R` option to specify those to extract.

■ **Example**

```
hex -R1000-1fff -R4000-41ff app.hzx
```

This will generate a single file, `app.hzx.bin`, which contains code or data which falls in the ranges 1000_{16} through $1FFF_{16}$ and 4000_{16} through $41FF_{16}$ from the file `app.hzx`.

You can combine these options with the `-W`, `-B`, and `-K` options to break the output into multiple files.

Index

Symbols

! operator, 45
& operator, 42
&& operator, 45
// comment, 27
:: operator
    defining public labels, 56
    retyping expressions, 46
; comment, 27
[] operator, 42
^ operator, 42
| operator, 42
|| operator, 45
~ operator, 42

A

ALIGN directive, 32
AND operator, 42, 45
array types
    assembler, defining, 38

B

binary constants, 24
bitwise operators, 42

C

-c option
    archiver (create archive), 80
    compiler driver, 20
comments
    assembler, 26
compiler driver
    summary of options, 70

D

-d option
    archiver (delete member), 81
-D option (define macro)
    compiler driver, 72
data
    aligning in assembler, 32
DB directive, 31
DD directive, 31
decimal constats, 24
DEFINED operator, 47
DL directive, 31
DW directive, 31

E

EQ operator, 44
EQU directive, 29
escape sequences
    in string constants, 26
.EXPORT directive, 55
exporting symbols
    from assembler files, 55

F

-F option (output format)
    in compiler driver, 71
FIELD directive, 36
.FILL directive, 33

G

GE operator, 44
GT operator, 44

H

HBYTE operator, 41
hexadecimal constants, 24
HIGH operator, 41
HWORD operator, 41

I

-I option (include directory)
    compiler driver, 71
images
    preparing for download, 86
    preparing for ROM, 87
.IMPORT directive, 56
importing symbols
    into assembler files, 56
INCLUDELIB directive, 57
index operator, 42

L

-L option (library directory)
    compiler driver, 73
    linker, 78
-l option (link library)
    compiler driver, 73
    linker, 78
LBYTE operator, 41
LE operator, 44
libraries
    including in assembler, 57
    linking on command line, 73
linker
    setting memory layout, 77
    summary of options, 76
LNOT operator, 45
LOW operator, 41
LT operator, 44
LWORD operator, 41

M

-M option (map file)
    linker, 78
macro definitions
    setting on command line, 72
memory layout
    using linker, 77

N

NE operator, 44
NOT operator, 42

O

object file lister
    summary of options, 84
octal constants, 24
OR operator, 42, 45
output formats
    hex extractor, 86
    linker, 77

P

pointer types
    in assembler, 39
PTR directive, 39

R

-r option
    archiver (replace member), 81
relational operators, 44

S

.SPACE directive, 33
string data
    defining, 32
STRUC directive, 36
structure types
    in assembler, 36

T

-t option
    archiver (table of contents), 80
    object lister (table of sections), 84
-T option (section address)
    linker, 77
THIS operator, 46

U

-U option (undefine macro)
    compiler driver, 72

V

-V option (show version)
    in compiler driver, 71

-v option (verbose)
compiler driver, 20, 71

X

XOR operator, 42