DLX Tools Guide

by David Knight
Adelaide University, South Australia
and Peter J. Ashenden
Ashenden Designs Pty. Ltd.

This document describes the DLX tools accompanying *The Designer's Guide to VHDL, Second Edition,* by Peter J. Ashenden. The software was developed by David Knight at Adelaide University, South Australia, and is Copyright Adelaide University. Permission is granted to use the software for educational and non-commercial use.

The DLX tools consist of the following programs:

- *dasm*—an assembler that generates relocatable or absolute object files,
- *dlnk*—a linker that combines separately assembled relocatable object files into a single relocatable object file and
- *dloc*—a locater that converts a relocatable object file into an absolute object file ready for loading and executing by a DLX processor.

These tools allow you to prepare programs for execution using the DLX instruction set interpreter models described in the case study in Chapter 15 of *The Designer's Guide to VHDL*.

The DLX Assembler

Invoking the assembler

The DLX assembler, called *dasm*, can perform both absolute and relocatable assembly. The general format of the command line is:

```
dasm [-alr] Filename
```

Absolute assembly

To perform an absolute assembly, type:

```
dasm -a [-l] Filename
```

Filename is the name of the file containing the DLX source program. By convention, the assembler expects the source filename to have the extension ".dls" ("DLX source file"). The assembler will generate an absolute object file with the same filename as the source file, but with extension ".dlx" ("DLX executable file"). The contents of this file are human-readable, so you can see what the assembler has produced. For example, to perform an absolute assembly of the program myprog (contained in the file myprog.dls), and generate a listing, type the command:

```
dasm -a -l myprog
```

The assembler will produce an executable file named myprog.dlx.

Relocatable assembly

To perform a relocatable assembly, type:

```
dasm -r [-l] Filename
```

Filename is the name of the file containing the DLX source program. The assembler expects the filename to have the extension ".dls" ("DLX source file"). The assembler will generate a relocatable object file with the same filename as the source file, but with extension ".dlo" ("DLX object file"). The contents of this file are human-readable, so you can see what the assembler has produced. For example, to perform a relocatable assembly of the program relprog (contained in the file relprog.dls), and generate a listing, type the command:

```
dasm -r -l relprog
```

The assembler will produce an executable file named relprog.dlo.

Generating a listing

The optional `-l` parameter instructs the assembler to generate a listing of your program and display it on the terminal. The listing file shows exactly how the assembler has translated the program, and a symbol table showing the name and value of each symbol. To save the listing in a file, you can use file redirection, for example:

```
dasm -a -l myprog > myprog.lst
```

The assembler listing will be saved in file myprog.lst

Source file format

Assembly code is traditionally typed in four columns:

```
Label  Opcode Operands  ;Comments
```

The label column is used to label statements. A label is optional, and may be followed by a colon sign (`:`), if desired, but this is ignored by the assembler. The opcode contains the instruction operation code, sometimes called a *mnemonic*, or the name of an assembler directive. The operands column contains any operands needed by the instruction or directive. The comments column is used to describe what is going on. Every comment must be preceded by a semicolon character (`;`).

Assembler directives

The dasm assembler has a number of directives for controlling the assembly process. These are described in the following sections.

.abs

```
        .abs
```

Tells the assembler that the following statements are to generate object code in the absolute segment. A label is not permitted. Operands are not permitted.

.align

```
        .align  k
```

Tells the assembler to insert space into the current segment so that the value of the location counter is a multiple of 2^k. If the location counter is already an exact multiple of 2^k, no space is inserted. A label is not permitted.

.ascii

```
Label  .ascii  "Characters terminated by a zero byte\0"
```

Tells the assembler to allocate storage for a sequence of characters, and to initialise the storage to contain the characters given. See "Characters and strings" below, for special characters.

.byte

```
Label  .byte  E1, E2, E3, …
```

Tells the assembler to allocate one byte of storage to hold a signed (2's-complement) value. The value stored in the byte is given by the expression E1. Multiple bytes may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expression must lie in the range –128 to 127, or an error will be reported.

.byteu

```
Label  .byteu  E1, E2, E3, …
```

Tells the assembler to allocate one byte of storage to hold an unsigned value. The value stored in the byte is given by the expression E1. Multiple bytes may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expression must lie in the range 0 to 255, or an error will be reported.

.equ

```
Label  .equ  E
```

Tells the assembler to enter the name Label into the symbol table, and assign it the value of the expression E. It must be possible to determine the value of E during the first pass of the assembly, or an error will be reported. Once defined, the value of the symbol cannot be changed.

.export

```
        .export  L1, L2, L3, …
```

Tells the assembler that the label L1 is to be exported for use in other programs. Multiple symbols can be imported by separating their names with commas. A label is not permitted. A definition for the labels L1, L2, …, must appear in this file.

.half

```
Label  .half  E1, E2, E3, …
```

Tells the assembler to allocate two bytes of storage (one halfword) to hold a signed (2's-complement) value. The value stored in the halfword is given by the expression E1. Multiple halfwords may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expression must lie in the range –32768 to 32767, or an error will be reported. If the location counter is not aligned on a halfword boundary, an error will be reported

.halfu

```
Label  .halfu  E1, E2, E3, …
```

Tells the assembler to allocate two bytes of storage (one halfword) to hold an unsigned value. The value stored in the halfword is given by the expression E1. Multiple halfwords may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expression must lie in the range 0 to 65535, or an error will be reported. If the location counter is not aligned on a halfword boundary, an error will be reported

.import

```
        .import  L1, L2, L3, …
```

Tells the assembler that the label L1 is to be imported from another program, and will not be defined in this file. Multiple symbols can be imported by separating their names with commas. A label is not permitted.

.org

```
        .org  E
```

Tells the assembler to set the location counter for the current segment to the value of the expression E. A label is not permitted. It must be possible to evaluate the expression E during the first pass of the assembler, or an error will be reported.

.seg

```
.seg  Name
```

Tells the assembler that the following statements are to generate object code in a relocatable segment called Name. A label is not permitted.

.set

```
Label  .set  E
```

Tells the assembler to enter the name Label into the symbol table, and assign it the value of the expression E. It must be possible to determine the value of E during the first pass of the assembly, or an error will be reported. A symbol defined using `.set` can have its value reassigned during the assembly process.

.space

```
Label  .space  E
```

Tells the assembler to allocate a block of bytes of uninitialised storage. The size of the block of storage is given by the expression E. It must be possible to evaluate the expression E during the first pass of the assembly, or an error will be reported.

.start

```
.start  E
```

Tells the assembler to record in the object file that execution of the program is to begin at the address give by the expression E. A label is not permitted.

.word

```
Label  .word  E1, E2, E3, …
```

Tells the assembler to allocate four bytes of storage (one word) to hold a signed (2's-complement) value. The value stored in the word is given by the expression E1. Multiple words may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expression must lie in the range –2147483648 to 2147483647, or an error will be reported. If the location counter is not aligned on a word boundary, an error will be reported.

.wordu

```
Label  .wordu  E1, E2, E3, …
```

Tells the assembler to allocate four bytes of storage (one word) to hold an unsigned value. The value stored in the word is given by the expression E1. Multiple words may be allocated and initialised by specifying multiple expressions, separated by commas. The value of each expres-

sion must lie in the range 0 to 4294967295, or an error will be reported. If the location counter is not aligned on a word boundary, an error will be reported.

Number format

By default the assembler expects numbers to be in base 10. Thus in the instruction:

```
addi    r1,r0,234
```

the value 234 is in base 10. You can specify a different base (say 16) by prefixing the number with the base, like this:

```
addi    r1,r0,16#EA
```

This is the number EA in base 16, which equals the number 234 in base 10.

The assembler accepts base numbers from 2 to 36, so you could use base 2 if you wanted to, like this:

```
addi    r1,r0,2#11101010
```

This is the binary number 1110 1010, which equals EA (base 16) and 234 (base 10).

The assmebler also accepts embedded underscore characters, to make long numbers more reabable, for example:

```
addi    r1,r0,2#1110_1010
```

or

```
addi    r1,r0,2#11_10_10_10
```

However, an underscore must always appear *between* two digits (i.e., not at the front or back, nor two in a row).

You can even express numbers in truly bizarre bases. For example:

```
addi    r1,r0,23#12
```

specifies the number 12 in base 23, which equals 25 (base 10), or worse:

```
addi    r1,r0,23#MD
```

This is the number MD in base 23. Since M=22 and D=13, this is the number 22×23+13 = 519 (base 10).

The formal syntax for numbers is:

```
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Letter ::= A | B | C | … | Y | Z
ExtendedDigit ::= Digit | Letter
DecimalNumber ::= Digit ([_] Digit)*
BasedlNumber ::= ExtendedDigit ([_] ExtendedDigit)*
Number ::= DecimalNumber [# BasedNumber]
```

Clearly, the set of extended digits that are acceptable in a number depends on the base number preceding the # sign.

Characters and strings

Characters and character-strings are specified by enclosing the character(s) in double quotes. dasm recognises a number of special combinations to represent control characters, and so forth, using the Unix conventions:

\a	Represents alarm-bell (ASCII BEL, 16#07)
\b	Represents backspace (ASCII BS, 16#08)
\f	Represents form-feed (ASCII FF, 16#0C)
\n	Represents newline (ASCII LF, 16#0D)
\r	Represents return (ASCII CR, 16#0A)
\t	Represents horizontal-tab (ASCII HT, 16#09)
\v	Represents vertical-tab (ASCII VT, 16#0B)
\\	Represents a single backslash (ASCII \, 16#5C)
\"	Represents a double-quote (ASCII " 16#22)

Assembler expressions

The assembler can evaluate complex expressions at assembly time. To maintain accuracy, expressions are evaluated using 64-bit arithmetic. Operators in expressions are evaluated strictly left-to-right, with no operator precedence rules. Parentheses can be used to control expression evaluation order. All calculations are performed using modulo 2^{64} arithmetic. The available operators are shown in Figure 1.

FIGURE 1

Operation	*Syntax*	*Notes*
Add	A + B	B is added to A.
Subtract	A - B	B is subtracted from A.
Multiply	A * B	B is multiplied by A.
Divide	A / B	A divided by B. If B is zero, an error will be reported.
Remainder	A % B	The remainder from dividing A by B. If B is zero, an error will be reported.
Complement	! A	Complement all the bits of A.
Or	A | B	Bitwise logical-or of B and A.
And	A & B	Bitwise logical-and of B and A.
Exclusive-or	A ^ B	The bitwise logical-exclusive-or of B and A.
Left shift	A << B	A left-shifted B bit positions. Vacated bits are set to zero. If B is negative, behaves like >>.
Right shift	A >> B	A right-shifted B bit positions. Vacated bits copy bit 63. If B is negative, behaves like <<.

Operations available in dasm expressions.

Pseudo instructions

The dasm assembler recognises a number of pseudo-instructions. The following sections list these pseudo-instructions, and the equivalent DLX instructions generated.

DLX core pseudo-instructions

```
pseudo instruction      DLX equivalent
add    rk,rj            add    rk,rk,rj
addi   rj,Ksgn          addi   rj,rj,Ksgn
addu   rk,rj            addu   rk,rk,rj
addui  rj,Kuns          addui  rj,rj,Kuns
and    rk,rj            and    rk,rk,rj
andi   rj,Kuns          andi   rj,rj,Kuns

bf     ri,Ksgn          beqz   ri,Ksgn
bt     ri,Ksgn          bnez   ri,Ksgn

clr    rj               addi   rj,r0,0

mov    rk,rj            add    rk,r0,rj
movi   rj,Ksgn          addi   rj,r0,Ksgn
movui  rj,Kuns          addui  rj,r0,Kuns

neg    rk               sub    rk,r0,rk
neg    rk,rj            sub    rk,r0,rj

or     rk,rj            or     rk,rk,rj
ori    rj,Kuns          ori    rj,rj,Kuns
```

```
sla    rk,rj          sla    rk,rk,rj
slai   rj,Kuns        sla    rj,rj,Kuns
sll    rk,rj          sll    rk,rk,rj
slli   rj,Kuns        sll    rj,rj,Kuns
sra    rk,rj          sra    rk,rk,rj
srai   rj,Kuns        sra    rj,rj,Kuns
srl    rk,rj          sra    rk,rk,rj
srli   rj,Kuns        sra    rj,rj,Kuns
sub    rk,rj          sub    rk,rk,rj
subi   rj,Ksgn        subi   rj,rj,Ksgn
subu   rk,rj          subu   rk,rk,rj
subui  rj,Kuns        subui  rj,rj,Kuns

xor    rk,rj          xor    rk,rk,rj
xori   rj,Kuns        xori   rj,rj,Kuns
```

DLX floating-point-unit pseudo-instructions

```
pseudo instruction     DLX equivalent

addd   fk,fj           addd   fk,fk,fj
addf   fk,fj           addf   fk,fk,fj

div    fk,fj           div    fk,fk,fj
divi   fj,Ksgn         divi   fj,fj,Kuns
divd   fk,fj           divd   fk,fk,fj
divf   fk,fj           divf   fk,fk,fj
divu   fk,fj           divu   fk,fk,fj
divui  fj,Kuns         divui  fj,fj,Kuns

mult   fk,fj           mult   fk,fk,fj
multi  fj,Ksgn         multi  fj,fj,Ksgn
multd  fk,fj           multd  fk,fk,fj
multf  fk,fj           multf  fk,fk,fj
multu  fk,fj           multu  fk,fk,fj
multui fj,Kuns         multui fj,fj,Kuns

subd   fk,fj           subd   fk,fk,fj
subf   fk,fj           subf   fk,fk,fj
```

Listing file layout

The listing file produced as a result of assembling a program allows you to see exactly how the assembler has generated code for your program. A few typical lines are shown below:

```
Location   Contents   Label      Opcode   Operands      Comments

00000000   8C01000C   main       lw       r1,initial    ;sum= initial
00000004   AC01000C+              sw       sum,r1
00000008   8C02000C+              lw       r2,sum        ;r2= sum
0000000C   00000200   initial:    .word    16#200
                                  .seg     My_code
                                  .import var
00000000+  8C010200               lw       r1,initial    ;sum= Initial
00000004+  AC01000C+              sw       sum,r1
```

```
00000008+ 8C010000#           lw     r1,var        ;r1= var
0000000C  =4         sum:     .space 2+2
```

On each line of output, the first column shows the address in memory, the second column shows the generated code for the source statement, and the remainder of the line shows the source statement. If an assembler directive contains an expression (as in .space, above), the assembler shows the value of the expression. For relocatable assembly, the assembler marks addresses subject to relocation with a + or – sign. (See segment MyCode above). Where an instruction refers to an imported symbol, the assembler marks the reference with a # sign.

File Formats

Normally, the object files produced by assemblers and other tools are in a format which has been chosen to make further machine processing convenient and fast. This results in such files not being directly readable by humans. The file formats used by the DLX tools contain essentially the same information as a "real" object files, but have been made deliberately human-readable, so you can see what sort of information is recorded in the file. By convention, absolute object files have the file extension ".dlx" ("DLX eXecutable file"), and relocatable object files have the file extension ".dlo" ("DLX object file").

The formats are specified in modified BNF. The meaning of the meta-symbols is:

::=	Is defined to be
|	Choose one of
[ ]	Optional item
[ ]*	Zero or more occurrences of
[ ]+	One or more occurrences of

Absolute Object File Format

An absolute file consists of an absolute segment, followed by an optional start address:

```
Absolute_file ::= Absolute_segment    [Start]
```

An absolute segment has the key word ".abs", followed by a block of memory data.

```
Absolute_segment ::= .abs  [ Memory_data ]*
```

A block of memory data consists of a number of lines. Each line begins with an address, and is followed by a list of data bytes. By convention, there are no more than 16 data bytes per line.

```
Memory_data ::= Address [ Byte ]*
```

The starting address for execution of a program is specified by the keyword ".start" followed by an address.

```
Start ::= .start Address
```

An address is a hexadecimal number, with from 1 to 8 digits

```
Address ::= hexadecimal number
```

A byte is a hexadecimal number with 1 or 2 digits

```
Byte ::= hexadecimal number
```

Example

```
.abs
00000000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000010 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

```
00000020 20 21
00000064 44 45
.start 14
```

This file loads the bytes 00 to 21 into locations 00000000 to 00000021, and the bytes 44 and 45 into locations 00000064 and 00000065. When loaded, execution is to begin at location 14

Relocatable Object File Format

A relocatable file consists of a number of memory segments, followed by zero or more relocation fixup records, followed zero or more linker symbol definition records, followed by zero or more linker linkup records, followed by an optional start address:

```
Relocatable_file ::=
    [ Segment ]+  [ Fixup ]*  [ Symdef ]*  [ Linkup ]*  [Start]
```

A segment is either an absolute segment, or a relocatable segment. (Absolute segments are in the same format as used in absolute object files. See appendix C) Thus:

```
Segment ::= [ Absolute_segment  | Relocatable_segment ]
```

An absolute segment has the key word ".abs", followed by a block of memory data.

```
Absolute_segment ::= .abs  [ Memory_data ]*
```

A relocatable segment has the keyword ".seg", followed by the segment name and the size of the segment in bytes. This is followed by a block of memory data.

```
Relocatable_segment ::=
    .seg  Segment_name  Segment_size   [ Memory_data ]*
```

A segment name is a dollar sign ($) followed by an identifier. The rules for forming an identifier are essentially those of any reasonable programming language.

```
Segment_name ::= $ String of printable characters
```

The size of a segment is a number, in the same format as an address

```
Segment_size ::= Address
```

A block of memory data consists of a number of lines. Each line begins with an address, and is followed by a list of data bytes. By convention, there are no more than 16 data bytes per line.

```
Memory_data ::= Address [ Byte ]*
```

A fixup specifies a location in memory that needs to be altered as a result of relocating the program. A fixup consists of the keyword ".fixup", followed by an address expression indicating the location that needs to be fixed up, a keyword indicating the type and size of relocation needed, and an address expression that give the value to be put at that location.

```
Fixup ::=
    .fixup Relocatable_address
              Relocation_type  Relocatable_address
```

A relocatable address is specified as plus/minus the base address of a segment plus/minus an offset.

```
Relocatable_address ::=
    [ [+ | -] Segment_name ]  [+ | - ] Address
```

A relocation type specifies the size and type of a relocation operation. Each operation is specified by a keyword. "SgnWord" indicates that the target location is to contain a signed word value. "SgnHalf" indicates that the target location is to contain a signed half-word value. "SgnByte" indicates that that a signed byte is to be stored. Similarly, "UnsWord", "UnsHalf", and "UnsByte", specify unsigned values. "RelWord", RelHalf", and "RelByte", indicate that a signed relative address is to be storedin the target location. (This is mostly used for branch instructions.)"SgnJump" indicates that a long-range signed jump address is to be stored.

```
Relocation_type ::=
    SgnWord | SgnHalf | SgnByte
    | UnsWord | UnsHalf | UnsByte
    | RelWord | RelHalf | RelByte
    | SgnJump
```

A symdef specifies the location of a public symbol defined within this object file. It consists of the keyword ".symdef" followed by a public symbol name, followed by a relocatable address specifiying the location of the symbol.

```
Symdef ::= .symdef  Symbol_name  Relocatable_address
```

A symbol name is a hatch (#) sign followed by an identifier. The rules for forming an identifier are essentially those of any reasonable programming language.

```
Symbol_name ::= # String of printable characters
```

A linkup specifies a location in memory that needs to be altered as a result of linking the program. A linkup consists of the keyword ".linkup", followed by an address expression indicating the location that needs to be linked, a keyword indicating the type and size of relocation needed, and an address expression that refers to a public symbol.

```
Linkup ::=
    .fixup Relocatable_address
              Relocation_type Symbolic_reference
```

A symbolic reference is specified as plus/minus the base address of a public symbol plus/minus an offset.

```
Symbolic_reference ::= [ [+ | -] Symbol_name ]  [+ | - ] Address
```

The starting address for execution of a program is specified by the keyword ".start" followed by a relocatable address.

```
Start ::= .start Address_expression
```

An address is a hexadecimal number, with from 1 to 8 digits

```
Address ::= hexadecimal number
```

A byte is either a hexadecimal number with 1 or 2 digits, or the value "??" which represents an undefined value. (?? is used to detect overlapping segments during relocation.)

```
Byte ::= ?? | hexadecimal number
```

Example

```
.seg $Prog 20
00000000 8C 01 00 00 24 02 00 00 20 63 00 00 8C 44 00 00
00000010 00 64 18 20 28 21 00 01 14 20 FF F0 00 00 00 01
.seg $Data 18
00000000 00 00 00 05 00 00 00 01 00 00 00 03 00 00 00 05
00000010 00 00 00 07 00 00 00 ??
.symdef #Count 0+$Data
.symdef #Numbers 4+$Prog
.linkup 2+$Prog SgnHalf 0+#Wombat
.linkup 8+$Prog UnsWord 4+#Roo
.start 0+$Prog
```

In this file there are two segments, $Prog of length 20, and $Data of length 18. Note that the byte at offset 17 in segment $Data is undefined (??). There are two public symbol definitions: #Count at offset 0 in the $Data segment, and #Numbers at offset 4 in the $Prog segment. There are two references to symbols defined in some other file: At address 2 in the $Prog segment there is a signed-halfword reference to the symbol #Wombat, and at address 6 in the $Prog segment there is an unsigned-word reference to the symbol #Roo, offset by 2. Program execution is to begin at offset 0 in the $Prog segment.

The DLX Linker

Invoking the linker

The DLX linker, called dlnk, links multiple relocatable object files together, and generates a single relocatable object file as output, containing all the segments within the original files During linking, dlnk creates a public symbol table containing the names of all symbols defined in .symdef records within the input files. Using the public symbol table, it then processes .linkup records to resolve cross references between the input files.

The general format of the command line is:

```
dlnk -o ObjFile [-m] InFiles
```

ObjFile is the name of the file that dlnk is to create. If a file extension is not specified, dlnk will append the extension ".dlo" ("DLX object file"). *InFiles* is a list of one or more relocatable files that are to be linked. dlnk expects each input file to have the extension ".dlo" ("DLX object file")

Generating a linking map

The optional -m parameter instructs the linker to generate a linking map and display it on the terminal. The map shows exactly how the segments from the input files have been combined, and gives the names and locations of all public symbols defined within the input files. To save the map in a file (for example, *LinkMap.lst*), you can use file redirection, for example:

```
dlnk -o ObjFile -m InFiles >LinkMap.lst
```

Example

Suppose the file prog.dlo contains:

```
.seg $Prog 20
00000000 8C 01 00 00 24 02 00 00 20 63 00 00 8C 44 00 00
00000010 00 64 18 20 28 21 00 01 14 20 FF F0 00 00 00 01
.linkup 2+$Prog SgnHalf 0+#Count
.linkup 6+$Prog SgnHalf 0+#Numbers
.start 0+$Prog
```

and the file data.dlo contains:

```
.seg $Data 18
00000000 00 00 00 05 00 00 00 01 00 00 00 03 00 00 00 05
00000010 00 00 00 07 00 00 00 09
.symdef #Count 0+$Data
.symdef #Numbers 4+$Data
```

Then the effect of executing:

```
dlnk -o progdata -m prog data
```

is to create the file progdata.dlo, containing:

```
.seg $Prog 20
00000000 8C 01 00 00 24 02 00 00 20 63 00 00 8C 44 00 00
00000010 00 64 18 20 28 21 00 01 14 20 FF F0 00 00 00 01
.seg Data 18
00000000 00 00 00 05 00 00 00 01 00 00 00 03 00 00 00 05
00000010 00 00 00 07 00 00 00 09
.fixup 2+$Prog SgnHalf 0+$Data
.fixup 6+$Prog SgnHalf 4+$Data
.start 0+$Prog
```

and to print the following linking map on the terminal:

Segment Map

```
Segment   Start     Length    File
Prog      00000000  20        prog.dlo
Data      00000000  18        data.dlo
```

Public Symbol Table

```
Symbol    Value
Count     0+$Data
Numbers   4+$Data
```

The DLX Locater

Invoking the locater

The DLX locater, called dloc, maps a relocatable object file to an absolute executable file, by assigning each relocatable segment to an absolute memory address. The mapping of segments to absolute addresses is controlled via a relocation address file.

The general format of the command line is:

```
dloc ObjFile [-m] [-l LocFile] [-x AbsFile]
```

ObjFile is the name of the file containing the DLX object program. dloc expects the object filename to have the extension ".dlo" ("DLX object file"). *LocFile* is the name of the relocation address file. dloc expects the address file to have the extension ".dla" ("DLX address file"). If *LocFile* is not specified, dloc will attempt to open a file with same name as *ObjFile*, but with the extension replaced by ".dla". *AbsFile* is the name of the file to receive the absolute executable file generated by dloc. dloc will create this file with the extension ".dlx" ("DLX executable file") If *AbsFile* is not specified, dloc will attempt to create a file with same name as *ObjFile*, but with the extension replaced by ".dlx".

Generating a relocation map

The optional -m parameter instructs the locater to generate a relocation map and display it on the terminal. The map shows exactly how dloc has converted each segment to an absolute address. To save the map in a file (for example, *LocMap.lst*), you can use file redirection, for example:

```
dloc ObjFile -m >LocMap.lst
```

Relocation address file format

The format of relocatable files is specified in modified BNF. The meaning of the meta-symbols is:

[ ]+ One or more occurrences of

A relocation file consists of one or more relocations items.

```
Relocation_file ::= [Relocation_item]+
```

A relocation_item consists of a start address, and end address, and a lsit of segment names. Segments from the object file will be placed into memory in the order of the names in the segment name list. The memory available runs from start address to end address, inclusive.

```
Relocation_item ::= Start_Address  EndAddress  [ Segment_name ]+
```

A start address is an address. It specifies where segment loading is to begin.

```
Start_Address ::= Address
```

An end address is an address. It specifies the last location where a group of segments may be loaded.

```
End_Address ::= Address
```

An address is a hexadecimal number, with from 1 to 8 digits

```
Address ::= hexadecimal number
```

A byte is a hexadecimal number with 1 or 2 digits

```
Byte ::= hexadecimal number
```

A segment name is an identifier, possibly preceded by a dollar sign. The rules for forming an identifier are essentially those of any reasonable programming language.

```
Segment_name ::= [ $ ] String of printable characters
```

Example

Suppose the file demo.dlo contains:

```
.seg $My_code 20
00000000  8C 01 00 00 AC 01 00 00 8C 01 00 00 8C 02 00 00
00000010  00 22 18 20 AC 03 00 00 0B FF FF EC 00 00 02 00
.seg $My_data 8
00000000  ?? ?? ?? ?? 00 00 00 32
.fixup 2+$My_code SgnHalf 1C+$My_code
.fixup 6+$My_code SgnHalf 0+$My_data
.fixup A+$My_code SgnHalf 14+$My_data
.fixup E+$My_code SgnHalf 0+$My_data
.fixup 16+$My_code SgnHalf 0+$My_data
.start 0+$My_code
```

and the file demo.dla contains:

```
00002100  00002FFF  My_code
00000150  000001FF  My_Data
```

Then the effect of executing:

```
dloc demo -m
```

is to create the file demo.dlx, containing:

```
.abs
00000154  00 00 00 32
00002100  8C 01 21 1C AC 01 01 50 8C 01 01 54 8C 02 01 50
00002110  00 22 18 20 AC 03 01 50 0B FF FF EC 00 00 02 00
.start 2100
```

and to print the following relocation map on the terminal:

```
Segment   Length    Address
My_data   8         00000150
My_code   20        00002000
```