CHAPTER 6

USING THE LINKER

This chapter discusses the Whitesmiths **lnk80** linker and the details of how it works. It describes each linker command line option, as well as the format of object modules, and explains how to use the linker's many special features. This chapter also provides example linker command lines that show you how to perform some useful operations.

The linker combines relocatable object files, selectively loading from libraries of such files made with **lby**, to create an executable image for standalone execution or for input to other binary reformatters.

You can use the linker to create object images with multiple segments. **lnk80** will also allow the object image that it creates to have local symbol regions, so the same library can be loaded multiple times for different segments, and to provide more control over which symbols are exposed. On microcontroller architectures this feature is useful if your executable image must be loaded into several noncontiguous areas in memory.

Note: The terms "segment" and "section" refer to different entities and are carefully kept distinct throughout this chapter. A "section" is a contiguous subcomponent of an object module that the linker treats as indivisible.

The assembler creates up to four such sections in each object module: the program text section, the data section, the debug section (which holds debugging information) and the bss section. The linker combines input sections in various ways, but will not break one up. The linker then maps these combined input sections into output sections in the executable image using the command line options you specify.

A "segment" is a logically unified block of memory in the executable image. An example is the code segment, which contains the executable instructions.

For most applications, the "sections" in an object module that the linker accepts as input are equivalent to the

"segments" of the executable image that the linker generates as output.

Overview

You use the linker to build your executable program from a variety of modules. These modules can be the output of the C cross compiler, or can be generated from handwritten assembly language code. Some modules can be linked unconditionally, still others can be selected only as needed from function libraries. All input to the linker, regardless of its source, must be reduced to object modules, which are then combined to produce the program file.

The output of the linker can be in the same format as its input. This applies for all output from **lnk80** except for the creation of a multi-section object file (described below). Thus, a program can be built in several stages, possibly with special handling at some of the stages. It can be used to build freestanding programs such as system bootstraps and embedded applications. It can also be used to make object modules that are loaded one place in memory but are designed to execute somewhere else. For example, a data section in ROM to be copied into RAM at program startup can be linked to run at its actual target memory location. Pointers will be initialized and address references will be in place.

As a side effect of producing files that can be reprocessed, **lnk80** retains information in the final program file that can be quite useful. The symbol table, or list of external identifiers, is handy when debugging programs, and the utility **rel80** can be made to produce a readable list of symbols from an object file. There are also "streams" of relocation data, which provide information on how to generate correct memory references if the object is relocated (moved to a different memory location). Finally, each object module has in its header useful information such as section sizes.

In most cases, the final program file created by **lnk80** is structurally identical to the object module input to **lnk80.** The only difference is that the executable file is complete and contains everything that it needs to run. There are a variety of utilities which will take the executable file and convert it to a form required for execution in specific microcontroller environments. The linker itself can perform some conversions, if all that is required is for certain portions of the executable file to be stripped off and for sections to be relocated in a particular way. You can therefore create executable programs

using the linker that can be passed directly to a PROM programmer.

The linker works as follows:

- Command line options applying to all sections of the object being built control the overall processing of linking input objects. These options are referred to in this chapter as "global options."

- Additional command line options apply only to specific sections of the object being built. These options are referred to in this chapter as "section control options."

- There are four types of sections in the output object; text sections, data sections, debug sections, and bss sections. Each of these is described below. There often is more than one of each of these sections in the output object. Each object file input contributes to the current text, data, debug and bss section.

- Sections can be relocated to execute at arbitrary places in physical memory, or "stacked" on suitable storage boundaries one after the other.

- The final output of the linker is a header (which contains information about the bss section), followed by all the text sections, all the data sections, all debug sections, the symbol table and the relocation streams for all the text, data and debug sections. There may also be an additional debug symbol table, which contains information used for debugging purposes. Some components of the output may be suppressed by the use of appropriate command line options, as described below.

- Note that all sections are contiguous in the linker output file, even if they have been linked in such a way that they will not be contiguous in memory. If they are linked to run at specific places in memory other than their contiguous locations in the linker output file, you must use the **hex80** utility to put the sections in their appropriate physical locations. See Chapter 8, "Using the Programming Support Utilities," for information.

Linker Command Line Processing

The command line to the linker is, in effect, a small control language designed to give the user a great deal of power in directing the actions of the linker. The basic structure of the command line input is a series of global flags followed by an arbitrary number of command items. The global flags provide control of the overall link process, setting linker behavior that should be constant throughout the current linker invocation. A command item is either an explicit linker option or the name of an input file (which serves as an implicit directive to link in that file or, if it is a library, scan it and link in any required modules of the library).

An explicit linker option consists of an option keyword followed by any parameters that the option may require. The options fall into four groups: the first group (**+text, +data** and **+bss.** controls the creation of new sections and has parameters which are selected from the set of local flags; the second group (**+new, +pub** and **+pri**) controls name regions and takes no parameters; the third group (**+sptext, +spdata** and **+spbss**) is used to reserve space in a particular section and has a required long int parameter; the fourth (**+def**) is used to define symbols and aliases and takes one required parameter, a string of the form ident1=ident2, or a string of the form ident1=constant. A description of each of these command line options appears below.

The manner in which the linker relocates the various sections is controlled by the **+text, +data** and **+bss** options and their parameters. As a section is being built, the linker keeps track of its size (among other things). If the size of a current section is zero when a command to start a new section of that type is encountered, no new section is created; however, the parameters of the option become the new parameters used in building that section.

Passing Options from STDIN

You can pass **lnk80** its argument list from STDIN. **lnk80** reads STDIN automatically if you pass it no arguments on the command line. In addition, the first occurrence of - in the list of command line arguments directs **lnk80** to read STDIN at that point in the argument list. Further occurrences of - are ignored.

lnk80 links the <files> you specify in order. If a file is in sequential library format, it is searched once from

start to end. Only those library modules that define public symbols for which there are currently outstanding unsatisfied references are included. Therefore, sequential libraries must be carefully ordered, or rescanned, to ensure that all references are resolved. A directory controlled library may also be of use in this context.

Inserting comments in Linker commands

When passing arguments from STDIN, or from an indirect file, each input line may be ended by a comment, which must be prefixed by a # character. A full line may begin with a comment. If you have to use the # as a significant character, you can escape it, using the syntax **\#**.

Here is an example for an indirect link file:

```
#
# Link commands
#
-o result.80      # result filename
+h                # multisegment
+data -b0x8000    # start data address
+text -b0xe000    # start text address
crts.80           # startup object file
mod1.o mod2.o     # input object files
libi.80           # C library
libm.80           # machine library
```

Linker Command Line Options

The linker accepts the following command line options, each of which is described in detail below.

lnk80 -[a cb cd ct +dead d +h +map=* max## mk o* pc* ps# vg vs vt x#] <sections>

where <sections> is one or more occurrences of

-[a# +bss b## +data +def* f# m## +new n* o## +pri +pub r# +spbss## +spdata## +sptext## +text x#]
<files>

The options are described in terms of the two groups listed above; the global options that apply to each section, and the section control options that apply only to specific sections.

Global Command Line Options

The global command line options that the linker accepts are:

-a make all relocation items and all symbols absolute. **-a** is used to prerelocate code that will be linked in elsewhere, and is not to be subsequently re-relocated.

-cb suppress code output for the bss section. This sets the bss size to zero in the header.

-cd suppress code output for the data section. **-cd** is used with the **-ct** and **-cz** option to make a module that defines only symbol values (for specifying addresses in shared libraries, etc.).

-ct suppress code output for the text section.

+dead generate a list of "deadwood" symbols; those symbols that were defined but never referenced.

-d do not define **bss** symbols, and do not complain about undefined symbols. Use this option for partial links, where the output module is to be used as input to **lnk80.**

+h create object module with a Whitesmiths multisegment header.

+map=* produce map information for the program being built. If **=*** is present, the map information produced is sent to the file specified by ***;** otherwise it is sent to **STDOUT.**

-max## makes the maximum size of a section ## bytes. The default value is 65535 bytes. This value is used for size checking when sections are combined.

-mk suppress the output of "MARKs" into the symbol table. These MARKs are the names of the input file or the names of object modules loaded from libraries, which are put in the symbol table to allow the output of the **rel80** utility to be more useful as a cross-referencing tool.

-o* write output module to the file *. The default is **xeq.80.**

-ps# set paragraph shift to #, which implies that the number of bytes in a paragraph is **2**#.** The default

value is 0 (making paragraphs 1 byte long). For more information, see the section "Address Components" below.

-pc* symbol inquiry character map. There is no default string.

-vg output to STDOUT a list of the debug symbols.

-vs output to STDOUT a listing of the section descriptor tables.

-vt output to STDOUT a listing of symbols, each with its value and its flags included.

-x# specify placement in the output module of the input text and data sections. The 2-weighted bit of # routes text section input; the 1-weighted bit routes data section input. If either bit is set, the corresponding sections are put in the **text** section output. Otherwise, they go in the data section. The default value is 2.

Section Control Options

This section describes the section control options that control the structure of individual sections of the output module.

A group of options to control a specific section must begin with one of the following four options: **+bss, +data** or **+text.** One of these options must precede any group of options so that the linker can determine which sections the options that follow apply to.

+bss start a new bss section and build that section as directed by the options that follow.

+data start a new data section and build that section as directed by the options that follow.

+text start a new text section and build that section as directed by the options that follow.

A new section of the specified type will not actually be created if the last section of that type has a size of zero. However, the new options will be processed and will override the previous values.

Options that control code regions have limited usefulness on microcontroller architectures. These options are:

+new start a new region. A "region" is a user definable group of input object modules which may have both public and private portions. The private portions of a region are local to that region and may not access or be accessed by anything outside the region. By default, a new region is given public access.

+pub make the following portion of a given region public.

+pri make the following portion of a given region private.

The option controlling symbol definition and aliases is:

+def* defines new symbols to the linker. The string * must be of the form:

- ident1=ident2 where ident1 and ident2 are both valid identifiers. This form is used to define aliases. The symbol ident1 is defined as the alias for the symbol ident2 and goes in the symbol table as an external DEF. (A DEF is an entity defined by a given module.) If ident2 is not already in the symbol table, it is placed there as a REF. (A REF is an entity referred to by a given module.)

- =ident where ident is a valid identifier. This form is used to put the symbol **ident** into the symbol table as an undefined public reference, usually to force loading of selected modules from a library.

- ident= where ident is a valid identifier. This form ident= is used to add **ident** to the symbol table as a defined absolute symbol with a value of zero.

- ident=constant where ident is a valid identifier and constant is a valid constant expressed with the standard C language syntax. This form is used to add **ident** to the symbol table as a defined absolute symbol with a value equal to **constant**.

Note: for more information about DEFs and REFs, refer to the discussion of the **lord80** utility in Chapter 8, and to the section "MARKs, REFs and DEFs" below.

The following options are used to reserve space in a given section:

+spbss## reserves ## bytes of space at the current location in the current bss section.

+spdata## reserves ## bytes of space at the current location in the current data section.

+sptext## reserves ## bytes of space at the current location in the current text section.

The remaining section control options (which must be used in combination with **+text**, **+data** or **+bss** as described above) are:

-a# sets the section attribute flags to #.

-b## set the bias of the section to ##.

-f# fill the section up to the nearest **2**#** byte bound by appending NUL bytes to the section. The default is zero, which specifies no padding.

-m## make the maximum size of an individual section ## bytes. The default size is 65535 bytes.

-n* set the name of a given section to *. Section names have at most 14 characters; longer names are truncated.

-o## set the offset for the start of the section to ##. The default is to set the offset equal to the bias.

-r# set the section bias to the end of the previous section rounded up to the nearest **2**#** byte boundary so that there are at least # zeroes in the resulting value. The default is zero. This option is ignored if the **-b#** option is specified.

-x# same as the **-x#** global option. This allows the code redirection to be changed from its original value.

For each section type, all options (except for **-b##**) apply to subsequent sections of that type unless explicitly reset. So, for example, the first occurrence of **+text -o0** causes each new text section to start at offset zero.

The bss section is always assumed to follow the data section in all input files. Unless **-b##** is given to set the bss section offset, the bss section will be made to follow the data section in the output file as well. It is permissible for text and data sections to overlap, as far as **lnk80** is concerned; the target machine may or may not make sense of this situation (as with separate instruction and data spaces). The debug section is set to start at zero. This behavior cannot be modified.

Object Formats

This section describes the object module format that the linker accepts as input, as well as the two possible object formats it can generate as output; standard object format and multisection object format. It also describes how the linker builds both standard and multisection objects.

Input Object Module Format

In general, most of the object modules that are input to the linker are created by the assembler. As the assembler processes its input, it directs its output code to one of four code sections: the text section, which normally receives all executable instructions and is normally read only; the data section, which normally receives all initialized data areas and is normally read/write; the bss section, which can accept only space reservations and is initialized to zeros at start of execution and the debug section, which receives debugging information when debugging is turned on.

The text section, data section and debug section are mapped to their respective positions in the standard object module. The bss section is only represented by its size in the object module header and the presence, in the symbol table, of the symbols defined in this section. It is always relocated relative to the end of the data section, so its bias is equal to data bias plus the data size; this number is not explicitly recorded anywhere in the object module header. If an object module input to the linker has been partially linked previously, then the component pieces could be fairly arbitrarily relocated.

Standard Object Format

The Version 3.3 standard relocatable object image consists of a header followed by a text section, a data section, a debug section, the symbol table, relocation information, and the debug symbol table.

The header consists of an identification byte, the value of which is 0xc0, a configuration byte, a short unsigned int containing the number of symbol table bytes, and eight more unsigned ints that specify: the number of bytes of object code defined by the text section, the number of

bytes of object code defined by the data section, the number of bytes of object code defined by the debug section, the number of bytes needed by the bss section, the number of bytes needed for stack plus heap, the text section offset, the data section offset and the debug section offset (always zero in the current implementation), respectively. Following this is a short int specifying the number of debug symbols in the debug symbol table, a long specifying the size of the debug symbol table in bytes, and another long specifying the seek address of the debug symbol table.

Byte order and size of all ints in the header are determined by the configuration byte, which contains all information needed to fully represent the header and remaining information in the file.

The configuration byte contains all information needed to fully represent the header and remaining information in the file. Its value val defines the following fields: If (val & 07), then **((val & 07) << 1) + 1** is the number of characters in the symbol table name field, so that values [1, 8) provide for odd lengths in the range [3, 15]. If **!(val & 07)**, then the symbol table name field is of variable length and consists of a one-byte character count (0 - 127 allowed) followed by the name. If **(val & 010)**, then ints are four bytes; otherwise they are two bytes. If **(val & 020)**, then ints in the data segment are represented least significant byte first. Otherwise, they are represented most significant byte first. Byte order is assumed to be purely ascending or purely descending. If **(val & 040)**, then even byte boundaries are enforced by the hardware. If **(val & 0100)**, then the text segment byte order is reversed from that of the data segment. Otherwise text segment byte order is the same. If **(val & 0200)**, some or all relocation information has been suppressed in the file.

The text section is relocated relative to the text section offset given in the header (usually zero), while the data section is relocated relative to the data section offset (usually zero). The debug section is relocated relative to address zero. Unless you specify the options **+bss -b##**, the bss section is relocated relative to the end of the data section.

Section relocation information consists of three successive byte streams, one for the text section, one for the data section and one for the debug section each terminated by a zero control byte.

Control bytes in the range [1, 31] cause that many bytes in the corresponding segment to be skipped. Bytes in the range [32, 63] skip 32 bytes, plus 256 times the control

byte minus 32, plus the number of bytes specified by the following relocation byte. A control byte of 64 reverses the order of bytes in an int for all subsequent relocation items.

All other control bytes control relocation of the next byte, short or long int in the corresponding segment. If the 1-weighted bit is set in such a control byte, a change in load bias must then be subtracted from the int. The 2-weighted bit determines whether relocation applies to a short or a long int. The 2-weighted bit is interpreted as follows: 1) if the bit is not set, relocation applies to a short. 2) If the bit is set, then relocation applies to a long int. The value of the control byte right-shifted two places, minus 16, constitutes a "symbol code".

A symbol code of 47 is replaced by a code obtained from the byte or bytes following in the relocation stream. If the next byte is less than 128, then the symbol code is its value plus 47. Otherwise, the code is that byte minus 128, the difference times 256, plus 175 plus the value of the relocation byte following that one.

A symbol code of zero calls for no further relocation. A symbol code of 1 through N, where N is the number of segments, means that a change in bias for that segment must be added to the item (short or long int). If the **-d** option is not specified (and there therefore are no undefined symbols), then codes N+1 through 2N are delta values for paragraph relocation. Otherwise, other symbol codes call for the value of the symbol table entry indexed by the symbol code minus N to be added to the item.

Each symbol table entry consists of a value of type int, a flag byte, and a variable length name consisting of a character count byte (with value in the range 0-127) followed by the name. The linker will accept as input object modules whose symbol table has fixed length names padded with trailing NULs, but the output will always have variable length names. The name length is determined by the configuration byte.

Meaningful flag values are 0 for "undefined," 3 for "defined debug relative," 4 for "defined absolute," 5 for "defined text relative," 6 for "defined data relative," and 7 for "defined bss relative." If the symbol is to be globally known, 010 is added to the flag value. This distinguishes various MARKs (defined above) from DEFs and REFs (also defined above) used in the linking process. In addition, 020 is added if the symbol has been defined but never referenced, and 040 is added if the symbol has been referenced. If the symbol is an alias for another symbol, the value 0100 is added to the flag byte; the immediately following symbol is the one named by the alias symbol. If

a symbol still undefined after linking has a nonzero value, **lnk80** assigns the symbol a unique area in the bss segment, the length of which is specified by the value, and considers the symbol defined. This occurs only if you have not specified the **-d** option.

Each debug symbol table entry consists of a value of type int, a type byte, a flag byte, and a variable length name consisting of a character count byte (with value in the range 0-127) followed by a name. The linker will accept as input object modules whose debug symbol table has fixed length names padded with trailing NULs, but the output object module will always have variable length names. The configuration byte determines the name length.

Each type byte value for debug symbols refers to a label that the code generator embeds in the assembly language source. The debug symbol table will contain only these labels. Meaningful values for the debug symbol table type flag byte are: 1 for **.dentry**, 2 for **.dtable**, 4 for **.dquit**, 8 for **.dline**, 16 for **.dendfn**, 32 for **.dstart**, and 64 for **.dfile**.

Meaningful values for the debug symbol table flag byte are the same as those described above for the symbol table flag byte.

Linking Standard Objects

The new text section is built by concatenating the text sections of the input object modules in the order the linker encounters them. As each input text section is added to the output text section, it is adjusted to be relocated relative to the end portion of the output text section so far constructed. The first input object module encountered is relocated relative to a value (usually zero) that can be specified to the linker. The output data section is built in the same manner from the data section components of the input object modules. The output debug section is built in this manner also, by concatenating the debug sections in the input object modules in the order in which the linker encounters them. As each new input debug section is added to the output debug section, it is adjusted to be relocated relative to the end portion of the output debug section constructed so far. If you do not specify the **-dxdebug** option to the compiler driver, the debug section will be of zero length. The size of the output bss section is the sum of the sizes of the input bss sections.

MARKs, DEFs and REFs

The linker builds a new symbol table based on the symbol tables in the input object modules, but it is not a simple concatenation with adjustments. There are three basic type of symbols that the linker puts into its internal symbol table: MARKs, REFs and DEFs. MARKs are essentially symbols entered to provide the user with information in the final output symbol table that enhances the load map that **rel80** can provide; they are not entered as symbols that the linker can "look up" (so the same name can appear any number of times) and do not contribute to the link process. DEFs are symbols that are defined in the object module in which they occur. REFs are symbols that are referenced by the object module in which they occur, but are not defined there.

The linker also builds a debug symbol table based on the debug symbol tables in any of the input object modules. It builds the debug symbol table by concatenating the debug symbol tables of each input object module in the order it encounters them. If debugging is not enabled for any of input object module, the debug symbol table will be of zero length.

Incoming MARKs are retained in the linker's internal symbol table unless the **-mk** option to the linker was specified. An incoming REF is added to the symbol table as a REF if that symbol is not already entered in the symbol table; otherwise, it is ignored (that reference has already been satisfied by a DEF or the reference has already been noted). An incoming DEF is added to the symbol table as a DEF if that symbol is not already entered in the symbol table; its value is adjusted to reflect how the linker is relocating the input object module in which it occurred. If it is present as a REF, the entry is changed to a DEF and the symbol's adjusted value is entered in the symbol table entry. If it is present as a DEF, an error occurs (multiply defined symbol).

When the linker is processing a library, an object module in the library becomes an input object module to the linker only if it has at least one DEF which satisfies some outstanding REF in the linker's internal symbol table. Thus, the simplest use of **lnk80** is to combine two files and check that no unused references remain. The command:

```
lnk80 file1 file2
```

combines **file1** and **file2** to form an executable image (plus header, symbol table and relocation bits) that could be

loaded at location zero and executed.

The executable file created by the linker must have no REFs in its symbol table. Otherwise, the linker emits the error message "undefined symbol" and returns failure.

Multisection Object Format

The standard Version 3.3 object image is limited to one text section, one data section, one bss section and one debug section. As such, it is insufficient when constructing certain classes of programs intended to run in a segmented memory architecture. To overcome this limitation, **lnk80** also has the capability to produce multisection object modules (although it cannot accept them as input). The basic file structure of the multisection object module is much the same as the standard object module: a multisection header followed by all the sections themselves, followed by the symbol table, the debug symbol table, and then the relocation streams for each of the sections.

The linker still works in terms of one text, data, debug, and bss section at a time. That is, any one input file can contribute only to the current text, data, debug and bss sections; new versions of any or all of these may be started between input files. The section order in the output object module is: all the text sections, all the data sections, all the debug sections and all the bss sections, respectively. Within a given section group (text, data, debug or bss), individual sections are in the order that the linker sees them.

The first sixteen bytes of the global header deal with the total structure of the object module. They are a one-byte "magic number," the value of which is 0xc2, a configuration byte (identical to the configuration byte of the standard object module header, as described above), the header size in bytes (two bytes), the number of sections (two bytes), the number of symbols (two bytes), the number of debug symbols (two bytes), the size of the debug symbol table (four bytes), the seek offset of the debug symbol table (four bytes), the seek address of the symbol table (four bytes), and the size of the symbol table (four bytes). A seek address is an offset in bytes from the first byte of the file.

Following this global header are the section descriptors; one 40-byte descriptor for each section. The number of sections is determined from the global portion of the header, as described above. Each section descriptor consists of: one byte (currently unused) describing the section subtype, one byte describing section attributes

(described below), a 14-byte section name, the virtual bias (four bytes), the offset (four bytes), the seek address of the section data (four bytes), the size of the section (four bytes), the seek address of the relocation stream (four bytes), and the size of the relocation stream (four bytes), for a total of 40 bytes.

Note: If the seek address for any portion of the object module is equal to zero, then that data item is not present in the file. A non-initialized section should have a valid section data size, but the seek address for that data should be zero.

The section attributes are defined below. Note that these values are used only by the **toprom** utility in the current implementation.

0x01	executable	0x10	shared
0x02	writable	0x20	sticky
0x04	readable	0x40	absolute
0x08	not initialized		

The symbol table in the multisection object module is similar to the standard symbol table, except each entry has a new component: the section number, of type unsigned char, which identifies the section in which the symbol was defined. (This imposes a limit of 255 sections in an object module.) The section number for a REF is zero.

Linking Multisection Objects

The multisection object module is built by the linker in much the same way as the standard object module. However, commands to start new sections may be interspersed in the list of input files. When such a command occurs, the current section of the specified type is terminated, along with its relocation stream, and a new one is started. The new section is relocated according to the options accompanying the command. The linker concatenates all the text sections in the order in which they were encountered into one text portion of the object module. This is followed by all the data sections and one unique debug section, similarly ordered. Each of these sections is also represented by a section descriptor in the header, which contains information on how the section is relocated, its size, where it and its relocation stream are located in the output file, etc. There are also section descriptors for all bss sections created.

Section Relocation

The linker relocates each section in a virtual address space which then can be mapped onto the actual machine address space. This virtual address space may exactly overlay the machine's address space or may be placed elsewhere in the memory address space (possibly with some storage boundary restrictions). However, the linker concatenates all the sections together in the output file; it does not build a scattered memory image. For the case of a scattered memory image, the linker can relocate each section for the place in memory where it will actually reside, but it does not put it there itself. You use the **hex80** utility to transform the linker output to a form which allows you to place the sections throughout memory.

For example, to link two files for execution at location 0xE000, with the data at 0xF000, pass the linker the following command:

```
lnk80 +text -b 0xE000 +data -b 0xF000 file1 file2
```

Again, the header, symbol table, and relocation bits are retained for use by other utilities like **hex80** and **rel80.**

Address Arithmetic

The two most important parameters describing a section are its bias and its offset. In nonsegmented architectures like the **Z80** there is no distinction between bias and offset. The bias is the address of the location in memory (or virtual memory on some systems) where the section is relocated to run. The offset of a segment will equal the bias. In this case you must set only the bias. The linker sets the offset automatically.

In a segmented architecture, the bias is the address of the start of the section in question. The offset is the distance, in bytes, of the start of the section from the start of the segment in which it is a part.

The paragraph shift specified by the **-ps#** option gives a measure of the resolution of the processor register used to hold the bias value of a segment. If the value specified by the -ps# option is n, then the resolution is **2**n.** For example, the value of n is 4 for the **Hitachi HD64180** family of processors.

In segmented architectures like the **Hitachi 64180,** the fundamental relationship between the bias and the offset

is:

bias = (SR << PS) + offset

where **SR** is the actual value used in a segment register and **PS** is the paragraph shift value you specify with the **-ps#** option.

In nonsegmented architectures like the **Z80** both PS and SR are usually equal to zero, so the formula becomes:

bias = offset

Setting Bias and Offset

The bias and offset of a section are controlled by the **-b##** option and **-o##** option, and are further influenced by the **-r#** option and **-f#** option. The rules for dealing with these options are described below.

Setting the Bias

If the **-b##** option is a special value (zero or -1) then the bias is set to the end of the last section rounded up to the nearest **2**#** byte boundary, where **#** is specified by the **-r#** option. The default value of the **-r#** option is zero, which implies that no rounding is done.

If the **-b##** option is not a special value (it is something other than zero or -1), then the bias is set to **##**. Note that -1 is the default value for **-b##**.

Setting the Offset

If the **-b##** option is zero, the section offset is set to continue from where the last section left off.

Otherwise, if the **-o##** option is -1 (the default value), then the offset is set equal to the bias.

Otherwise, the offset is set to the value **##** specified by the **-o##** option.

Addresses have three components:

```
Bias      =   32-bit start address of the segment
Offset    =    16 or 32-bit offset within the segment
Paragraph =   16 or 32-bit component whose resolution is set by ps
```

where ps represents "paragraph shift" as set by the -ps# flag.

The relationship between these three quantities can be stated as follows:

```
Bias = (Paragraph << ps) + Offset
```

The values of symbols are their offsets.

There are methods for finding the paragraph of the corresponding segment for each symbol, as well as other relevant information. If _abc is a symbol, then B_abc is the paragraph number of the start of the segment in which _abc is defined. In addition, E_abc is the paragraph number of the first paragraph past the end of the segment, and X_abc is the size of the segment in paragraphs. The start of the segment in bytes, the first byte past the end of the segment, and the size in bytes are given by b_abc, e_abc, and x_abc, respectively. The six characters used in this mapping may be altered by the use of the -pc* flag.

The ps component in the equation above determines the paragraph size to be 2**ps. This value is set by using the -ps# flag to lnk. The default is -ps0. For an 8086, -ps4 gives 16-byte paragraphs. For a 68000, use -ps0, since paragraphs are irrelevant. One could use -ps16 for bank selection machines, or for a Z8000.

Completing the Section

Before the section is complete, it is zero-filled up to the nearest 2**# byte boundary, where # is specified by the **-f#** option. The default value of the **-f#** option is zero, which implies no filling.

The section ordering that is implied by referring to the previous, or last, section is the same as the section ordering of the output object module. The default values for the first section are a bias of zero and an offset of zero.

The default behavior of the linker is to build the output for a nonsegmented memory system starting at the zero address of the virtual address space. The relocation stream contains all the data necessary to make the adjustments required to relocate the virtual address space anywhere in the machine address space. If the starting location in the machine address space is known prior to link time, then that value can be used as the bias of the first section to link.

User Control of Stack and Heap

Should the user prefer to control the definition of the stack plus heap area, the best approach is to use a new bss section, or perhaps two, for separating the heap area from the stack. The section in which the stack is defined should be given the name "stack" to prevent the linker from doubly defining things. Then the **+spbss** command can be used to reserve space and the **+def** command can be used to define the necessary symbols. The linker always maintains the three internal symbols **__text__**, **__data__**, and **__bss__** as the current location in the current text, data, and bss sections respectively. These should be used in defining the required symbols. For example, the command **+def __stklo=__bss__** defines the symbol **__stklo** as the current location in the current bss section (presumably the one in which the stack is being defined). Then space should be reserved with **+spbss** and the symbol **__stkhi** defined.

Note: Defining the stack and heap in this manner should be done very carefully so that the rest of the program can actually access the user defined areas correctly.

Return Values

lnk80 returns success if no error messages are printed to STDOUT; that is, if no undefined symbols remain and if all reads and writes succeed. Otherwise it returns failure.

Special Topics

This section explains some special linker capabilities that may have limited applicability for building most kinds of microcontroller applications.

Private Name Regions

Private name regions are used when you wish to link together a group of files and expose only some to the symbol names that they define. In other words, all the DEFs other than those chosen for export become MARKs. This lets you link a larger program in groups without worrying about names intended only for local usage in one group colliding with identical names intended to be local to another group. Private name regions let you keep names truly local, so the problem of name space pollution is much more manageable.

An explicit use for private name regions in an 8086 environment is in building a large program as a set of "islands" of code compiled for the small model. Only those functions and external objects that need be shared among islands are explicitly declared as **far**. With this scheme, each island is linked with its own set of near-callable machine-support functions (and any other library functions that may safely be replicated). To avoid complaints when multiple copies of the same file redefine symbols, each such contribution is placed in a private name region accessible only to other files in the same island.

The basic sequence of commands for each island looks like:

+new <public files> **+pri** <private libraries>

Any symbols defined in <public files> are known outside this private name region. Any symbols defined in <private libraries> are known only within this region; hence they may safely be redefined as private to other regions as well.

You can also use this machinery to rename symbols and simultaneously hide the old name. If, for example, you have a file that defines **open**, and you wish to rename it **gr_open**, just type:

```
lnk80 -o new -d +def gr_open=open +pri old
```

This creates the file **new**, which is similar to **old** except that **open** is now a private symbol, which disappears in the file **new**, and **gr_open** is now a DEF, the value of which matches the value of **open**. The global option **-d** calls for a partial link, so the output file may be used as subsequent input to the linker.

Note: All symbols defined in a private region are local symbols and will not appear in the symbol table of the output file.

Renaming Symbols

At times it may be desirable to provide a symbol with an alias and to hide the original name (i.e., to prevent its definition from being used by the linker as a DEF which satisfies REFs to that symbol name). As an example, suppose that the function **func** in the C library provided with the compiler does not do everything that is desired of it for some special application. There are three methods of handling this situation (we will ignore the alternative of trying to live with the existing function's deficiencies).

The first method is to write a new version of the function that performs as required and link it into the program being built before linking in the libraries. This will cause the new definition of **func** to satisfy any references to that function, so the linker does not include the version from the library because it is not needed. This method has two major drawbacks: first, a new function must be written and debugged to provide something which basically already exists; second, the details of exactly what the function must do and how it must do it may not be available, thus preventing a proper implementation of the function.

The second approach is to write a new function, say **myfunc**, which does the extra processing required and then calls the standard function **func**. This approach will generally work, unless the original function **func** is called by other functions in the libraries. In that case, the extra function behavior cannot occur when **func** is called from library functions, since it is actually **myfunc** that performs it.

The third approach is to use the aliasing capabilities of the linker. Like the second method, a new function will be written which performs the new behavior and then calls the old function. The twist is to give the old function a new name and hide its old name. Then the new function is given the old function's name and, when it calls the old function, it uses the new name, or alias, for that function. The following linker script provides a specific example of this technique for the function **func:**

```
line 1   -o progname -b 0x1000
line 2   +text +data -o0 +bss -b0
line 3   +new
line 4   Crts.xx
line 5   +def _oldfunc=_func
line 6   +pri func.o
line 7   +new
line 8   prog.o newfunc.o
line 9   <libraries>
```

Note: The function name **func** as referenced here is the name as seen by the C programmer. The name which is used in the linker for purposes of aliasing is the name as seen at the object module level. The name transformation from the C level to the object module level may be as simple as prepending a leading underscore (or some other character), or may involve a more complex transformation. For more information, see the section "C Interface to Assembly Language" in Chapter 3.

The main thing to note here is that both **func.o** and **newfunc.o** both define (different) functions named **func.** The second function **func** defined in **newfunc.o** calls the old **func** function by its alias **oldfunc.**

Name regions provide limited scope control for symbol names. The **+new** command starts a new name region, which will be in effect until the next **+new** command. Within a region there are public and private name spaces. These are entered by the **+pub** and **+pri** commands; by default, **+new** starts in the public name space.

Lines 1,2 are the basic linker commands for setting up a separate I/D program. Note that there may be other options required here, either by the system itself or by the user.

Line 3 starts a new region, initially in the public name space.

Line 4 specifies the startup code for the system being used.

Line 5 establishes the symbol **_oldfunc** as an alias for the symbol **_func**. The symbol **_oldfunc** is entered in the symbol table as a public definition. The symbol **_func** is entered as a private reference in the current region.

Line 6 switches to the private name space in the current region. Then **func.o** is linked and provides a definition (private, of course) which satisfies the reference to **_func**.

Line 7 starts a new name region, which is in the public name space by default. Now no reference to the symbol **_func** can reach the definition created on Line 6. That definition can only be reached now by using the symbol **_oldfunc**, which is publicly defined as an alias for it.

Line 8 links the user program and the module **newfunc.o**, which provides a new (and public) definition of **_func**. In this module the old version is accessed by its alias. This new version will satisfy all references to **_func** made in **prog.o** and the libraries.

Line 9 links in the required libraries.

The rules governing which name space a symbol belongs to are as follows:

- Any symbol definition in the public space is public and satisfies all outstanding and future references to that symbol.

- Any symbol definition in the private space of the current region is private and will satisfy any private reference in the current region.

- All private definitions of a symbol must occur before a public definition of that symbol. After a public definition of a symbol, any other definition of that symbol will cause a "multiply defined symbol" error.

- Any number of private definitions are allowed, but each must be in a separate region to prevent a multiply defined symbol error.

- Any new reference is associated with the region in which the reference is made. It can be satisfied by a private definition in that region, or by a public definition. A previous definition of that symbol will satisfy the reference if that definition is public, or if the definition is private and the reference is made in the same region as the definition.

- If a new reference to a symbol occurs, and that symbol still has an outstanding unsatisfied reference made in another region, then that symbol is marked as requiring a public definition to satisfy it.

- Any definition of a symbol must satisfy all outstanding references to that symbol; therefore, a private definition of a symbol which requires a public definition causes a blocked symbol reference error.

- No symbol reference can "reach" any definition made earlier than the most recent definition.

Example Linker Command Lines

This section shows you how to use the linker to perform some basic operations.

To create an **Z80** file from **file.o:**

```
lnk80 -o xeq +text -b0x2000 /c/lib/crts.80 file.o
    /c/lib/libi.z80 /c/lib/libm.z80 +def__memory=__bss__
```