White Paper

Using the Z8 Encore!® Linker

WP000701-1103

ZiLOG Worldwide Headquarters • 532 Race Street • San Jose, CA 95126
Telephone: 408.558.8500 • Fax: 408.558.8300 • www.zilog.com

Abstract

The ZiLOG Z8 Encore! Linker takes object files and libraries as input and links them together to generate an executable load file. This document provides an overview of the functionality of the linker and how it can be used along with the compiler/assembler provided in ZDSII to generate an output load file which can be downloaded into the target system. Please refer to the ZDSII User's Manual wherever details of the linker command syntax are needed.

The linker performs two major functions:

- Links together object modules and libraries into one executable load file by resolving external references.
- Locates the relocatable segments and assigns absolute addresses to them.

ZiLOG microcontrollers, including the Z8 Encore!, support multiple address spaces. The ZiLOG Xtools C compiler, assembler and linker together provide a mechanism to use these address spaces. The compiler provides language extensions to declare variables in different address spaces. The assembler provides a means to associate a segment with an address space, and the linker has the ability to link multiple segments associated with an address space and locate them.

Z8 Encore! Microcontroller Address Spaces

The Z8 Encore! microcontroller has two address spaces:

1. Rom: This represents the processor flash. The maximum range for this address space is 0-FFFF.
2. Register File: This represents the processor RAM. The maximum range for this address space if 0-FFF.

The Register File can be addressed using 4, 8 and 12 bit addresses by using instructions with appropriate addressing modes.

Z8 Encore! Assembler Address Spaces

The Z8 Encore! assembler provides three address spaces:

1. Rom: This corresponds to the micro controller address space Rom.
2. RData: This represents the first page of Register File which is 8 bit addressable.
3. EData: This represents the entire Register File which is 12 bit addressable.

Both RData and EData together make the Register File address space of the microcontroller.

In Z8 Encore! assembly, users can associate segments with any of the above three address spaces.

The Z8 Encore! assembler provides the following pre-defined segments:

- code : The address space is Rom. Contains program code.
- near_bss: The address space is RData. Contains uninitialized data.
- near_data: The address space is RData. Contains initialized data.
- far_bss: The address space is EData. Contains uninitialized data.
- far_data: The address space is EData. Contains initialized data.
- text: The default address space is RData. Contains constant data.
- rom_text: The address space is Rom. Contains constant data.
- __vectors: The address space is Rom. Contains the interrupt vector table.

Note the distinction between the segments "code", which is given to executable code, and "text", used for constant data. We point this out because some commonly used object file formats use the term "text" to indicate executable code.

Z8 Encore! Compiler Address Spaces

The Z8 Encore! compiler provides language extensions to place variables in different address spaces. These language extensions are provided through storage class specifiers which can be used on individual data objects similar to the const and volatile keywords in the ANSI standard. The storage class specifiers provided by the compiler are:

1. rom : The variable is to be allocated in a segment with Rom address space.
2. near : The variable is to be allocated in a segment with RData address space.
3. far : The variable is to be allocated in a segment with EData address space.

Some examples are:

```
near int nvar;
far int fvar;
rom int rvar;
```

The compiler places variables into assembler segments as follows:

- fname_text: Used for executable code, where fname is the function to which the code belongs. The address space is Rom.
- near_bss: Un-initialized global and static variables with near storage specifiers are assigned to this segment. The address space is RData.

- near_data: Initialized globals and statics with near storage specifiers are assigned to this segment. The address space is RData.
- far_bss: Un-initialized globals and statics with far storage specifiers are assigned to this segment. The address space is EData.
- far_data: Initialized globals and statics with far storage specifiers are assigned to this segment. The address space is EData.
- text: Globals and statics with the const storage qualifier are assigned to this segment. This segment is later mapped to RData, EData or Rom address space based on the compilation memory model selected and the option whether const is placed in ram or rom. The compiler memory models are discussed in the *Using the ZiLOG Xtools Z8 Encore! C Compiler* white paper (WP0006), available in the Xtools ZDS II distribution.
- rom_text: The global and statics with rom storage specifiers are assigned to this segment. The address space is Rom.
- __vectors: All the interrupt definitions, defined using the C SET_VECTOR directive, are assigned to this segment.

The globals and statics for which the storage class specifier is not given by the user are assigned to a default space by the compiler. The default space is decided based on the memory model (Large or Small) selected by the user for the compilation. The memory model also dictates whether local variables and function parameters would be located in RData or in EData.

Linking Z8 Encore! Applications

The Z8 Encore! linker is used to link compiled and assembled object module files, compiler libraries, user created libraries, and special object module files used for C runtime initializations. The details of how these files are linked are controlled by the commands given in the linker command file (LCF).

The default LCF is automatically generated by ZDSII IDE whenever a build command is issued. It has information about the ranges of various address spaces for the selected device, the assignment of segments to spaces, order of linking, etc. The default LCF can be overridden by the user as explained below.

The linker processes the object modules (in the order in which they are specified in the linker command file), resolves the external references between the modules, and then locates the segments into the appropriate address spaces as per the LCF.

The linker depicts the memory of the microcontroller using a hierarchical memory model containing spaces and segments. The various memory regions of a microcontroller are associated with spaces. Multiple segments can belong to a given space. Each space has a range associated with it that identifies valid addresses

for that space. The hierarchical memory model for Z8 Encore! is shown in Figure 1. Figure 2 depicts how the linker links and locates segments in different object modules.

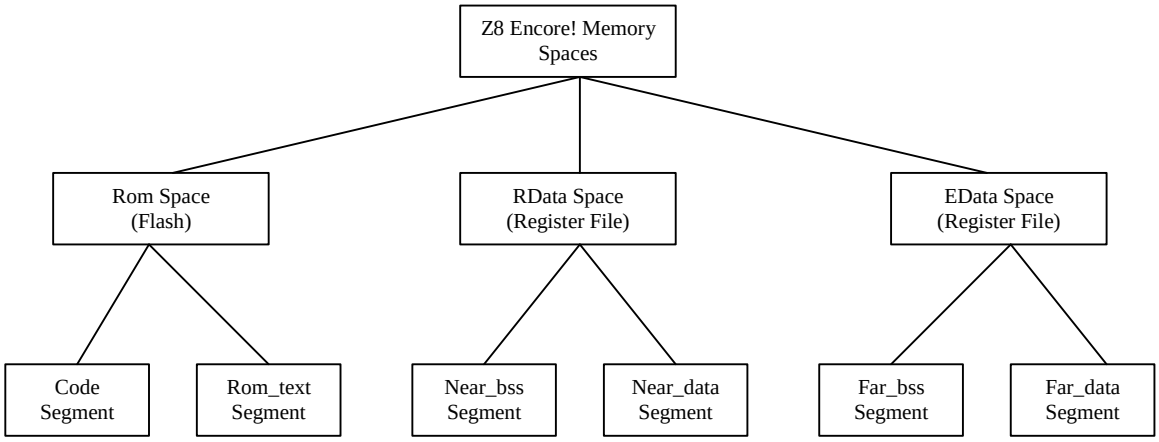

Figure 1. Z8 Encore! Hierarchical Memory Model

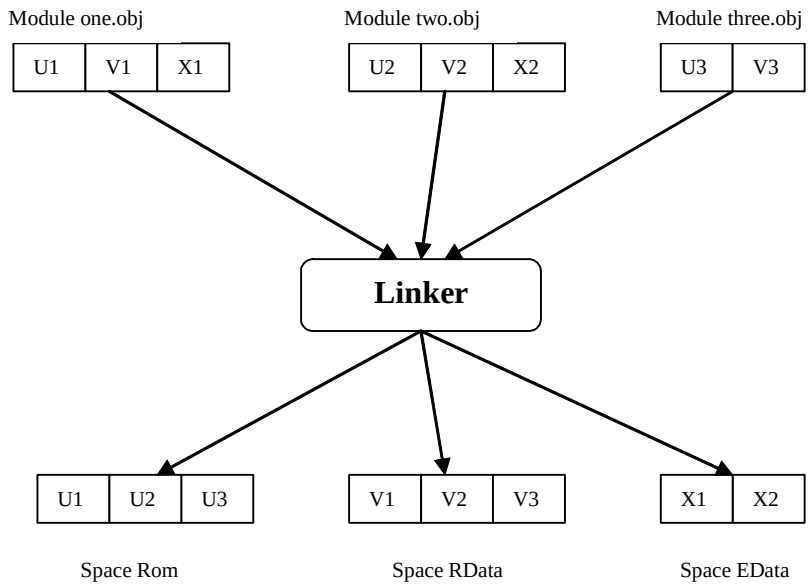

Figure 2. Multiple File Linking

Linker Referenced Files

The default LCF generated by the ZDSII IDE references system object files and libraries based on the compilation memory model selected by the user. A list of the system object files and libraries is given in the Table 1. The LCF automatically selects and links to the appropriate version of the C run time and (if necessary) floating point libraries, from the list shown in Table 1, based on the user's project settings.

Table 1. Linker Referenced Files

File	Description
Startups.obj	C Startup for small model.
Startupl.obj	C Startup for large model.
Fpdumys.obj	Floating point do nothing stubs for small model.
Fpdumyl.obj	Floating point do nothing stubs for large model.
Crtld.lib	C Run time library for large dynamic model, no debug information.
Crtldd.lib	C Run time library for large dynamic model, with debug information.
Fpld.lib	Floating point library for large dynamic model, no debug information.
Fpldd.lib	Floating point library for large dynamic model, with debug information.
Crtls.lib	C Run time library for large static model, no debug information.
Crtlsd.lib	C Run time library for large static model, with debug information.
Fpls.lib	Floating point library for large static model, no debug information.
Fplsd.lib	Floating point library for large static model, with debug information.
Crtsd.lib	C Run time library for small dynamic model, no debug information.
Crtsdd.lib	C Run time library for small dynamic model, with debug information.
Fpsd.lib	Floating point library for small dynamic model, no debug information.
Fpsdd.lib	Floating point library for small dynamic model, with debug information.
Crtss.lib	C Run time library for small static model, no debug information.
Crtssd.lib	C Run time library for small static model, with debug information.
Fpss.lib	Floating point library for small static model, no debug information.
Fpssd.lib	Floating point library for small static model, with debug information.

Linker Symbols

The default LCF defines some system symbols, which are used by the C startup file to initialize the stack pointer, clear the un-initialized variables to zero, set the initialized variables to their initial value, set the heap base, etc. Table 2 below

shows the list of symbols that may be defined in the LCF, depending on the compilation memory selected by the user.

Table 2. Linker Symbols

Symbol	Description
_low_neardata	Base of near_data segment after linking
_len_neardata	Length of near_data segment after linking
_low_near_romdata	Base of the rom copy of near_data segment after linking
_low_fardata	Base of far_data segment after linking
_len_fardata	Length of far_data segment after linking
_low_far_romdata	Base of the rom copy of far_data segment after linking
_low_nearbss	Base of near_bss segment after linking
_len_nearbss	Length of near_bss segment after linking
_low_farbss	Base of far_bss segment after linking
_len_farbss	Length of far_bss segment after linking
_far_stack	Top of stack for large model is set as highest address of EData.
_near_stack	Top of stack for small model is set as highest address of RData.
_far_heapbot	Base of heap for large model is set as highest allocated EData address
_near_heapbot	Base of heap for small model is set as highest allocated RData address

A Sample Linker Command File

The sample default LCF for large dynamic compilation model is discussed here as a good example of the contents of an LCF in practice and how the linker commands it contains work to configure the user's load file. The default LCF is automatically generated by the ZDSII IDE. If the project name is test.pro, for example, the default LCF name is test.lnk. The user can add additional directives to linking process by specifying them under **Project -> Settings -> Linker -> Input -> Add Directives.** The user can alternatively define his own LCF by using **Project -> Settings -> Linker -> Input -> Use Existing** or **Project -> Settings -> Linker -> Input -> Custom** option.

The most important of the linker commands and options in the default LCF will now be discussed individually, in the order in which they would typically be found:

```
-FORMAT=OMF695
-NOigcase -map -quiet -warn -NOwarnoverlap -NOxref
```

In this command, the linker output file format is selected to be OMF695 which is based on the IEEE 695 object file format. This setting is generated from options

selected in **Project -> Settings -> Linker -> Output**. The other options shown here are all generated from the settings selected in **Project -> Settings -> Linker -> General**.

```
RANGE ROM $0 : $FFFF
RANGE RDATA $20 : $FF
RANGE EDATA $100 : $EFF
```

The ranges for the three address spaces are defined here. These ranges are taken from the settings in **Project -> Settings -> Target -> Memory**.

```
CHANGE TEXT=EDATA
CHANGE TEXT=FAR_DATA
```

The TEXT section is assigned to EDATA address space by the above command. Recall that the TEXT section contains the C const variables and can be remapped into any of the three available spaces (Rdata, Edata, or Rom) based on project settings. In this example, the Large model is in effect and so the TEXT segment will be assigned either to the Edata or Rom space, depending on whether RAM or ROM, respectively, is selected in **Project -> Settings -> C -> Code Generation -> Const Variable Placement**. The second command adds the const data of TEXT to the list FAR_DATA that must be initialized from ROM by the initialization code.

```
ORDER FAR_BSS, FAR_DATA
ORDER NEAR_BSS,NEAR_DATA
```

These ORDER commands specify the link order of these segments. The FAR_BSS segment will be placed at lower addresses with the FAR_DATA segment immediately following it in the EData space. Similarly, NEAR_DATA will follow after NEAR_BSS in RData space.

```
COPY NEAR_DATA ROM
COPY FAR_DATA ROM
```

This COPY command is a linker directive to make the linker place a copy of the initialized data segments NEAR_DATA and FAR_DATA into the ROM address space. At run time, the C startup module will then copy the initialized data back from the ROM address space to the RData and EData address spaces. This is the standard method to ensure that variables get their required initialization from a non-volatile stored copy, in a typical embedded application where there is no offline memory such as disk storage from which initialized variables can be loaded.

```
define _low_near_romdata = copy base of NEAR_DATA
define _low_neardata = base of NEAR_DATA
define _len_neardata = length of NEAR_DATA
define _low_far_romdata = copy base of FAR_DATA
define _low_fardata = base of FAR_DATA
define _len_fardata = length of FAR_DATA
define _low_nearbss = base of NEAR_BSS
```

```
define _len_nearbss = length of NEAR_BSS
define _low_farbss = base of FAR_BSS
define _len_farbss = length of FAR_BSS
define _far_heaptop = highaddr of EDATA
define _far_stack = highaddr of EDATA
define _near_stack = highaddr of RDATA
define _far_heapbot = top of EDATA
```

These are the linker symbol definitions described in Table 2. They allow the compiler to know the bounds of the different memory areas that must be initialized in different ways by the C startup module.

```
"c:\sample\test"= \
 C:\PROGRA~1\ZiLOG\ZD3E4C~1.0\lib\startupL.obj, \
 .\foo.obj, \
 C:\PROGRA~1\ZiLOG\ZD3E4C~1.0\lib\crtLDD.lib, \
 C:\PROGRA~1\ZiLOG\ZD3E4C~1.0\lib\fpLDD.lib
```

This final command shows that in this example, the linker output file will be named test.lod. The source object file (foo.obj) is to be linked with the other modules that will be required to make a complete executable load file. In this case, those other modules are the C startup modules for the large model (startupl.obj), the C run time library for the large model with dynamic frames (crtldd.lib) and the floating point library for that same configuration (fpldd.lib).

An important point to understand in using the linker is that the linker intelligently links in only those functions from a given module that are necessary to resolve its list of unresolved symbols. For example, while the C run time library contains a very large number of functions from the C Standard Library, if your application only calls two of those functions then only those two will be linked into your application (plus any functions that are called by those two functions in turn). This means it's safe for the user to simply link in a large library or module, like crtLDD.lib and fpLDD.lib in this example. The user doesn't have to worry about bloat caused by unnecessary code being linked in, and doesn't have to do the extra work of painstakingly finding the unresolved symbols for themselves and linking only to those specific functions.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Customer Support Center
532 Race Street
San Jose, CA 95126
USA
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer