Application Note

Mapping Data Objects to Z8 Encore!® Flash Memory

AN013803-0708

Z8 Encore!® Flash Microcontrollers

Abstract

This Application Note describes three techniques for significantly minimizing the use of data RAM in Zilog's Z8 Encore!® microcontroller unit (MCU) when developing applications within the Zilog Developer Studio II—Integrated Development Environment (ZDS II—IDE).

Discussion

For any processor, it is advisable to preserve as much of free data RAM space as possible, especially when using the small memory model for efficiency.

Normally at download, initialized variables are stored along with all code contained in Flash memory. At reset, the required data RAM space is allocated to the variables and initialized to the set value in the code. Therefore, during run time, initialized variables occupy space in Flash memory and in data RAM. This process also occurs for constant definitions.

Mapping several initialized variables and constant definitions to a specified variable in data RAM considerably reduces data RAM usage. Such a mapping technique (hereafter referred to as the Flash Mapping technique) is possible with all of the data types supported by ZDS II.

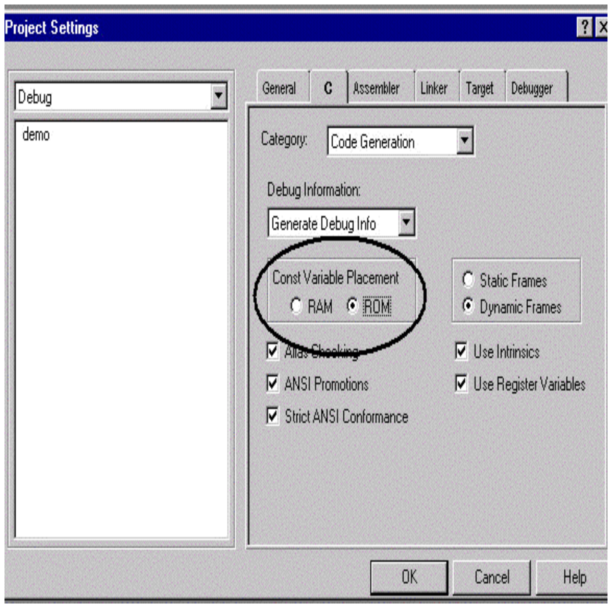
Listed below are three ways to use the Flash Mapping technique within the ZDS II programming environment:

- Using the ZDS II Graphical User Interface (GUI)
- Inside a C routine
- Inside an Assembly routine

Mapping Data Objects Using the ZDS II GUI

Note: *The following description refers to ZDS II v4.1.0.*

The ZDS II IDE provides a direct way to map all of the constant definitions in a project to Flash memory. In the **Project Settings** dialog box (see Figure 1), click the **C** tab. Select **Code Generation** from the **Category** drop down list.



Category: Code Generation

Debug Information: Generate Debug Info

Const Variable Placement: RAM [✓ ROM]

Static Frames [✓ Dynamic Frames]

- [✓] Alias Checking
- [✓] ANSI Promotions
- [✓] Strict ANSI Conformance
- [✓] Use Intrinsics
- [✓] Use Register Variables

Figure 1. Flash Mapping Technique Using ZDS II GUI

The **Const Variable Placement** option facilitates a choice of memory space (RAM and ROM or only ROM) for the constant definition, *const*. The ZDS II default selects the RAM radio button, as a result of the *const* definitions are located in Flash and RAM. To prevent RAM usage, the ROM radio button must be selected to force the compiler to restrict all of the *const* definitions to Flash memory.

However, there are some limitations to this mapping technique, as listed below.

- It is restricted to *const* type-qualified objects (definitions) and excludes all initialized variables. Thus, modification of Flash-mapped data objects is not possible.
- Selecting the ROM button subjects all the constant definitions within the scope of the project to be confined to Flash memory.

Mapping Data Objects for a C Routine

To circumvent the limitations of the ZDS II mapping technique, the compiler directive `ROM` can be used. This directive facilitates:

1. Modifications of Flash-mapped data objects.
2. Selective placement of data objects into Flash memory.

The syntax for the `ROM` directive is:

```
ROM data_type identifier
definition
```

For example, consider the following declarations:

```
ROM char example_array1[] =
"Welcome to Z8 Encore!";

char example_array2[] = "8-Bit
Flash Microcontroller";
```

The string `Welcome to Z8 Encore!` is stored only in Flash memory, while the string `8-Bit Flash Microcontroller` is not only stored in Flash memory (at download) but also duplicated in the RAM space (at Reset).

In a commercial application, there are certain considerations to be noted. The standard C library functions expect their operands to be in the RAM space. The presence of these definitions in Flash implies that the library functions cannot be used to access the definitions. It is therefore necessary to write a small routine that transfers the data from Flash to RAM whenever it is required.

Copying is specific to data type, so the length of the data to be transferred must be supplied along with source and destination addresses. This instance is true for all data types except for the *string* data type, as it is limited by the NULL character.

The C-routine provided below copies string data from Flash to RAM.

```
void rom_to_data_copy(char
*dest,char ROM *src)
{
  do
  {
  *dest++ = *src++;
  }while (*src);
}
```

Note: *The above routine is required only if the C library functions are used. Alternatively, specific functions can be written to access the data directly from Flash memory.*

The following code offers a simple illustration of how the `ROM` directive and the `rom_to_data_copy()` routines can be used together to save data RAM space.

```
#include <stdio.h>
ROM char fmt1[] = "Hello World
\n";// Store first string in
// ROM
ROM char fmt2[] = "------------
\n";// Store second string
// in ROM
char fmt[20];// Variable to
//hold either string at run-
// time
```

```
void rom_to_data_copy(char
*dest, char ROM *src)
{
  do
  {
  *dest++ = *src++;
  }while (*src);
}

void main(void)
{
  rom_to_data_copy(fmt,fmt2);
  //Load the second string to
  //RAM
  printf(fmt);// This prints
  //fmt2
  rom_to_data_copy(fmt,fmt1);
  //Load the first string to RAM
  printf(fmt);// This prints
  //fmt1
  rom_to_data_copy(fmt,fmt2);
  // Load the second string to
  //RAM
  printf(fmt); // This prints
  //fmt2
}
```

In the above example, the Flash-mapped data objects are not type-qualified to be *const*; therefore, their modification is possible via the compiler. However, the data resides in Flash memory, and updating it involves programming Flash memory. The details of programming Flash memory are outside the scope of this Application Note. For implementing a Flash Programmer using Z8 Encore!, see Reference on page 4.

Mapping Data Objects for an Assembly Routine

To restrict the mapping of data objects to Flash memory for an Assembly program, the DEFINE assembler directive is used along with the SPACE clause (that is, the word SPACE is typed into the routine). The necessary data objects are collected under one user-defined segment. This is achieved with the DEFINE directive with the following syntax.

```
DEFINE
<identifier>,[<SPACE_clause>],
[<ALIGN_clause>],<ORG_clause>]
```

In the above definition, a segment is defined with its associated address space (as defined by the SPACE clause), alignment, and origin. If a SPACE clause is not provided, the DEFINE directive uses the defaults of the current space. The ALIGN and ORG clauses are optional to the purpose of this Application Note.

To avoid the duplication of data objects, it is necessary to define this segment in the ZDS II predefined ROM space.

In an Assembly routine, the example_array1 (see Mapping Data Objects for a C Routine) can be declared as described below.

Note the use of the SPACE clause with the predefined memory space ROM to map the arrays.

```
DEFINE flash_data_here, SPACE =
ROM;

SEGMENT flash_data_here;

example_array1  DB  'Hello
World',0;  Flash String with
NULL
```

The example_array2 (see Mapping Data Objects for a C Routine), which is not a Flash mapped object, can be defined as follows:

```
SEGMENT near_data;

example_array2  DB -----------
',0;  RAM String with NULL
```

For an Assembly program, Flash data can be accessed directly through the LDC and LDCI instructions. Table 1 on page 4 lists the syntax for the LDC and LDCI instructions.

Table 1. LDC and LDCI Instruction Syntax

Instruction	LDC	LDCI
Definition	**L**oa**D** **C**onstant	**L**oa**D** **C**onstant auto-**I**ncrement
Syntax	LDC <*dst*>,<*src*>	LDCI <*dst*>,<*src*>
Operands	<r>,<Irr>	<r>,<Irr>
	<Irr>,<r>	<Irr>,<Ir>
Example	LDC @R2,@RR6	LDCI @R2,@RR6

The LDC instruction loads a byte constant from Program Memory into a working register. The address of the Program Memory location is specified by a working register pair. The contents of the source operand are unaffected.

The LDCI instruction performs a similar operation to the LDC instruction; however, both source and destination addresses in the working registers are incremented automatically after every transfer. The contents of the source operand are unaffected. For more information on these instructions, see Reference.

The assembly code listed below can be used by you for copying the data objects mapped to Flash memory into data RAM. Note how the LDC instruction is used.

```
XDEF rom_to_data_copy;

rom_to_data_copy :
LD R3, 3(R15)
LD R0, 4(R15)
LD R1, 5(R15)
LDC R2, @RR0 ; Use LDC for Flash
;read
LD @R3, R2
LD R0, 3(R15)
INC R0
LD 3(R15), R0
LD R1, 4(R15)
LD R0, 5(R15)
CLR R2
ADD R0, #%01
ADC R1, R2
LD 4(R15), R1
LD 5(R15), R0
LD R0, 4(R15)
LD R1, 5(R15)
LDC R2, @RR0 ;Use LDC for Flash
;read
OR R2, R2
JR NZ, rom_to_data_copy
RET
```

While updating data in Flash, a Flash programming utility is required. While a discussion of a Flash programming utility is outside the scope of this Application Note, a reference to a C-language implementation of the Flash Programmer using Z8 Encore! is provided in Reference.

Summary

This Application Note describes Flash Mapping techniques to be used within the ZDS II programming environment. The economical use of the data RAM space makes these techniques ideal while using a small memory model. Implementation details are described to help both the Assembly programmer and the C-programmer.

Although Flash mapping is most effective when used with constants and variables that are seldom changed, there is no restriction on using them for variables that are modified frequently. Access issues related to Flash memory are discussed and solutions are presented for C and Assembly programs.

Reference

The documents associated with Z8 Encore! available on www.zilog.com are provided below:

- eZ8 CPU User Manual (UM0128)
- Zilog Developer Studio II—Z8 Encore! User Manual (UM0130)
- Flash Loader Utility for the Z8 Encore! XP MCU Application Note (AN0118)

Warning: DO NOT USE IN LIFE SUPPORT