



Flash Programming on the EB402 Application Note



Trademark Acknowledgments

© VeriSilicon Holdings Co., Ltd. All rights reserved worldwide. VeriSilicon, the VeriSilicon logo, ZSP and the ZSP logo are the trademarks of VeriSilicon Holdings Co., Ltd. in the United States and/or other jurisdictions. All other trademarks are the property of their respective holders.

Printed in P.R.China.

VeriSilicon Holdings Co., Ltd. reserves all its copy rights and other intellectual property rights, ownership, powers, benefits and rights arising or to arise from this manual. All or part of the contents of this manual may be changed by VeriSilicon Microelectronics (Shanghai) Co., Ltd. without notice at any time for any reason, including but not limited to improvement of the product relating hereto.

VeriSilicon Holdings Co., Ltd. shall not undertake or assume any obligation, responsibility or liability arising out of or in respect of the application or use of the product described herein, except for reasonable, careful and normal uses.

Nothing, whether in whole or in part, within this manual can be reproduced, duplicated, copied, changed or disposed of in any form or by any means without prior written consent by VeriSilicon Holdings Co., Ltd.



Contents

1. Introduction	4
2. Flash Configuration	5
3. Jumper Settings	5
4. Applications with Serial Debugging Support	6
4.1 Write Program to Reside in Flash.....	6
4.2 Develop Code to Reprogram External Flash	6
4.3 Linker Script File.....	7
4.4 Compile and Run Application using Proper Settings	9
5. System Examples	9
5.1 Restoring Factory Default Code into Flash.....	10
5.2 User Application in Flash with No Serial Debugging ..	11
5.3 Integrating Serial Debug Host with User Application Code	12
6. Main Program and Link to Files	15

1. Introduction

This document is a reference for users who will be programming the on-board flash memory for serial debugging and developing of the VSI402Z and VSI402ZX on the EB402 Evaluation Board. This document assumes the user is familiar with the EB402 and the JTAG debug environment.

This information is meant to assist:

- Engineers who are utilizing the flash memory on the EB402 to design, debug and develop DSP applications
- Engineers developing applications which boot from external flash memory and execute from internal memory.

The EB402 is the evaluation board for the VSI402ZX digital signal processor (DSP) device. The EB402 provides a hardware platform for evaluating the device and a software platform for developing, debugging, and demonstrating real-time applications for the VSI402ZX DSP. The EB402 consists of internal memory mapped inside the VSI402ZX, and on-board external memory.

This application note examines the on-board external flash memory, the process necessary to boot the DSP device from this memory, and the reprogramming of flash for debugging and development purposes. Specific hardware and software requirements must be executed to perform this task. A detailed description, step-by-step processes, and example codes are included. The topics covered in this document are:

- Flash Configuration
- Hardware Settings
- Serial Debugging Applications
- Generic Programming Applications
- Integrating Serial Debug into Existing Code

While the contents of the application note relate to the EB402, the principles of using external memory for boot code can be applied to a stand-alone system.

2. Flash Configuration

The flash on the EB402 Evaluation Board is a Micron MT28F800B3WG-10T. This is a 512 Kword x 16 bit device with 100 ns access time, connected to the VSI402ZX through the XBUS. The flash is normally connected to chip select ICS0N, so for all examples in this app note, the flash is accessed through this chip select. ICS0N is the only chip select used for instruction fetches to external memory. This chip select is accessed by using page 0 in the memory page control register (mempcr), and external instruction memory is selected by using the dir, sis, and lis bits in the %smode control register. [Table 1](#) shows the bits and their definitions.

Table 1 Memory Control Bits in %smode Register

Bit	Description
dir	disable internal instruction memory
ddr	disable internal data memory
lis	load from instruction memory
sis	store to instruction space
fie	force internal execution

3. Jumper Settings

This section describes the jumper settings on the EB402 that control flash operation. [Table 2](#) describes the jumpers that affect flash behavior.

Table 2 Jumper Settings

Ref Des	Pins	Function
JP3	–	Open to boot from ROM Install jumper to boot from flash
JP15	–	Open to enable instruction block
JP17	–	Open to enable flash as instruction memory
JP18	2-3	Select flash memory as XBUS boot device
JP20	–	Install to provide flash programming voltage

The A1 8 jumper, JP1 5, selects the high-order memory space of the flash. With JP15 installed, ADDR18 is pulled-down and the memory is configured as four 64 Kword blocks. When JP15 is uninstalled, flash is configured as three 64 Kword blocks, two 4 Kword parameter blocks and one 8 Kword boot block. This design allows you to have two versions of the code in the flash memory, with each version in a separate memory location accessible by installing or removing A18 (JP15).

4. Applications with Serial Debugging Support

The ZSP SDK tools provide debugging support through either a JTAG interface or through an RS-232 interface into the EB402. The run-time code to support serial-port debugging does not fit into the internal boot ROM for the VSI402ZX and is instead programmed into the Flash on the EB402. To perform serial-based debugging, the EB402 must boot and execute from the external flash.

There are five key steps to developing code for the flash on the EB402. They are:

1. Write program to reside in flash
2. Develop program to program external flash (in this case from internal memory)
3. Develop linker application to locate code internally/externally at the appropriate addresses
4. Compile and run application with proper jumper settings.
5. Add jumper JP3 to boot from flash and run new application.

4.1 Write Program to Reside in Flash

Writing a flash-resident program is very similar to writing code used with the debugger with a few exceptions. Normally the debugger/ROM code handles the startup function of the processor after leaving reset. In flash applications, the user is responsible for writing the startup code which resides at 0xF800. This code would normally setup the stack, initialize any required hardware, and branch to the main application. Since the reset vector is only eight words long, code located at 0xF800 normally branches to the real startup code followed by a branch to the main application.

4.2 Develop Code to Reprogram External Flash

The application used to reprogram the external flash generally consists of three main parts:

1. Configuring the memory interface of the VSI402ZX (chip selects, asynchronous mode, wait states, memory map)
2. Sending control signals to the flash to erase memory
3. Reprogramming the flash with the new code

[Section 6, "Main Program and Link to Files,"](#) lists a C program which performs these steps. The example erases all of the flash memory during the erase function and programs a block of data starting at flash address

0xF800 for 0x800 words. In more sophisticated applications, the flash reprogramming application might program different blocks of various sizes. Examples include the interrupt vector table at 0xF800, main program code at a location in lower memory, or a copy utility to move the program from flash to internal instruction memory and then begin executing from internal memory. Organization of the various sections and sizes depends on the system requirements.

4.3 Linker Script File

A linker script file is required when building the programming application to properly locate sections of code. Example 1 below defines four sections: the interrupt vector table, the text section, the initialized data section, and the code to be written to flash.

Example 1 Linker Script File

```
SECTIONS {  
    isr_table 0x0000 : { * (int_vect) }  
    .text 0x4000 : {*(.text) }  
    .data 0x6000 : {*(.data) }  
    _flash_loadaddr = 0x2000;  
    flash_boot_section 0x8000 : AT (_flash_loadaddr)  
    {*(fboot)}  
}
```

The `AT` syntax in the section information instructs the linker to resolve link information in the code at one location, while physically placing it into another part of memory. In the example, the linker resolves addresses for the code located at 0x8000, while it actually loads it in memory at address 0x2000. This method is similar to that used for code overlays, where code is linked at one location and loaded at another.

The symbol `flash_load_addr` in the link file allows use of a calculated address, such as end of `.text` section (`0x4000 + SIZEOF(.text)`). Using arithmetic in the linker script file aids development because the programmer does not have to predetermine locations for code to an exact number. [Figure 1](#) shows an illustration of the internal and resulting external memory map associated with the linker file shown in Example 2.

Example 2 Linker Script File with Calculated Offsets

```
SECTIONS {
    isr_table 0x0 000 : { * (intvect) }
    .text 0x4000: {*(.text) }
    .data 0x6000: {*(.data) }
    _flash_loadaddr = 0x2000;
    flash_boot_section 0xf800 : AT (_flash_loadaddr)
    { * (fboot)} flash_end = (0xf800 + SIZEOF(flash
    _boot _section));
    flash_main = (_flash_loadaddr + SIZEOF(flash _boot
    _section));
    flash_2 _section (flash_end) : AT (flash_main) { *
    (new_user_code)
}
}
```

The code in Example 2 links two different sections of code into external memory: `flash_boot_section` and `flash_2_section`. The second section (`flash_2_section`) is both linked and loaded immediately after the `flash_boot_section` by using the `SIZEOF` operator to calculate the location for the code.

In the source code, the boot section has the name `fboot` and the application has the name `new_user_code`. These names must be included into the source code with the syntax

```
.section "fboot", "ax"
```

and

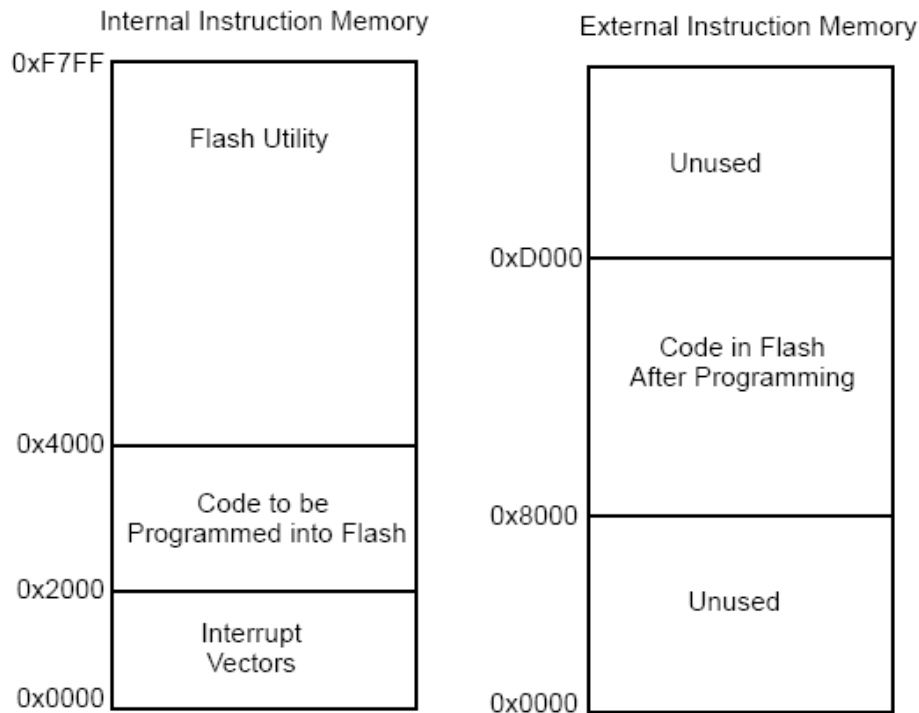
```
.section "new_user_code", "ax"
```

Note: The quotation marks are required.

The `ax` syntax tells the linker this code is part of instruction memory and will be linked into that memory block.

Figure 1 Internal and External Instruction Memory

Internal Instruction Memory External Instruction Memory
0xF7FF



4.4 Compile and Run Application using Proper Settings

Table 2 lists the jumper settings required to program the flash. Normally, the application boots from ROM (JP3 open) which allows JTAG software-based debugging to load the programming application which reprograms the flash. JP20 provides the required voltage to the flash pins for programming activity and must be installed.

4.5 Run New User Application from Flash

To boot and execute from flash, install jumper JP3 and reset or cycle power to the board. This causes the board to use external flash for boot space (0xF800–0xFFFF) and runs the new application loaded into flash.

5. System Examples

This section contains descriptions of a number of system examples. The code for the examples is located on the VeriSilicon website at <http://www.verisilicon.com>

Section 6, “Main Program and Link to Files,” page 15 contains a main program in C for reprogramming the flash. The examples in this section are:

- Restoring factory default code into flash
- Programming a new application into flash with no serial debugging
- Integrating the serial debug monitor with new application code

- Booting from flash and executing from internal memory

5.1 Restoring Factory Default Code into Flash

This example contains the files required to reprogram the flash to the factory default value. This value will allow use of the serial interface to the evaluation board for using the ZSP SDK debugger.

5.1.1 File Descriptions

This subsection describes the files that make up the example.

`Main.C` is the main program used to reprogram the flash. It contains code to configure the memories (wait states, asynchronous mode, chip selects), install interrupts to allow debugging, erase and reprogram the flash, then scroll the LEDs in a pattern to indicate either success or failure.

`Flash.C` contains a number of functions used to erase, program, and verify flash. These routines are specific to the Micron flash used on the EB402, but could easily be modified to match another flash memory.

`Utils.C` is a collection of simple utilities to allow memory reads/writes to various sections of memory (internal/external instruction/data). These utilities can be called from C and illustrate the required bit settings in configuration registers for the various memory accesses. This file also contains some simple timer routines used to delay for a specified amount of time.

`Intvects.S` is a file with interrupt vectors used in the main application. These interrupt vectors allow debugging while reprogramming the flash from either internal instruction memory (JTAG-based debugging) or from external instruction memory (serial-based debugging).

`Xboot_400LV.S` is the code to be programmed into flash. It contains startup code, JTAG or UART-based debugging support, and HPI boot capability. It allows the ZSP SDK debugger to operate through the serial port on a PC instead of requiring a JTAG interface board.

`Boot2.lcf` is the linker script file used to build the application and locate code properly.

5.1.2 Building and Running the Application

To run the application, create a new project in `sdlaunch` and add all of the source files listed above. Under Project Options->Linker, specify `boot2.lcf` as an additional linker option. Verify proper jumper settings. Build the application and run with a target of either JTAG-SW debugging, or Serial

debugging. Run the program to completion.

If the programming is successful, the LEDs on the EB402 will scroll from one end to the other, then delay and repeat the pattern. If programming fails, the LEDs will flash from one pattern to another. Failures are generally due to an invalid jumper setting on the board.

After running this application, the flash should contain valid code to allow serial-based debugging from a host. Install JP3 to boot from flash instead of ROM. Specify the Remote setting in `sdlaunch` under

Project Options->Debugging to use serial-based debugging for any other projects.

5.2 User Application in Flash with No Serial Debugging

This application is an example of installing user code into flash, then booting and executing that code from flash. In this instance, no serial debug support is provided—the user application code runs independently of any other system.

5.2.1 Description of Files

This subsection describes the files that make up the example.

`User1.S` is the new user application which is linked at 0xF800 and executes when the application boots from flash. The first section of code is the interrupt vector table, located at 0xF800. Only the reset handler is populated with a real interrupt routine; all other handlers branch to a generic interrupt handler.

The generic interrupt handler and the reset handler immediately follow the interrupt vector table. The reset handler performs five tasks:

1. Set the PIO lines as inputs.
2. Set the external memory interface to asynchronous for all memory accesses.
3. Set the wait states for the external signals.
4. Set up the stack pointer in register `r12`. The utility defines a symbol `bootrom_compiler_stack_location`, but this could be passed in from the linker as an external variable.
5. Calls the user's main application. In this case, the main program also resides above 0xF800 since it is small. Examples of far

calls or loading to internal memory are shown in other examples.

The main program flashes the LEDs in an incrementing pattern to provide a visual indication of successful operation.

5.2.2 Building and Running the Application

Build the application as described in [Section 5.1, “Restoring Factory Default Code into Flash,”](#) page 9. When the application executes, it increments a counter and displays the count value on the LEDs.

5.3 Integrating Serial Debug Host with User Application Code

In some instances, a developer might want to reprogram the flash and execute the new target application in flash while still running the debugger through the serial port. This example illustrates how to integrate a new end-application with the host debug software in flash.

5.3.1 Description of Files

This subsection describes the files that make up the example.

`User3.S` is a user application which increments a counter and displays the results on the LEDs. In this example, the program is in a stand alone file and is linked in with the serial debug software during the build process. The LED pattern is inverted from that in [Section 5.2, “User Application in Flash with No Serial Debugging.”](#)

`Xboot_400LV_User.S` is the serial debug code to be programmed into flash. It contains code nearly identical to the factory default, except it calls the user application program after startup instead of continuing in an infinite loop writing to the LEDs. All other features are identical to those described in [Section 5.1, “Restoring Factory Default Code into Flash,”](#) so the UART is initialized and ready to support serial-based debugging after startup.

`Boot3.lcf` is the linker script file. It uses arithmetic to determine the proper link and load addresses for the new user application, which is placed immediately after `Xboot_400LV_User.S`.

5.3.2 Building and Running the Application

Building and running the application is similar to [Section 5.1, “Restoring Factory Default Code into Flash.”](#) When the new code executes from flash, it increments a counter and displays the value on the LEDs, with the

bits inverted relative to the example in [Section 5.2, “User Application in Flash with No Serial Debugging.”](#)

5.4 Programming/Booting from Flash and Internal Memory Execution

One common use of flash is to support booting and a code move to internal memory before executing the actual program. Executing from internal instruction memory is faster than using external flash and uses less power. This example shows how to program the flash, copy the user code from flash to internal RAM, and execute the code from internal memory.

5.4.1 Description of Files

This subsection describes the files that make up the example.

`Main.c` is similar to the previous versions, but a second block of flash is programmed (0x0000–0x1FFF) in addition to the boot block (0xF800–0xFFFF).

`Boot_app.S` is nearly identical to `Xboot_400LV_User.S` from [Section 5.3, “Integrating Serial Debug Host with User Application Code.”](#) `Boot_app.S` contains serial debug support, JTAG debug support, and a startup section. After booting, it calls `copy_app`.

`Copy_app.S` is the code used to copy the application from flash to internal instruction memory. In this example, it does a 1:1 copy (moving 0x2000 words from 0x0000 flash to 0x0000 instruction memory). It could relocate the code or some other required function, perhaps including using a different external page for flash. Two methods are illustrated in the file, the first using direct load/store commands, and the second using DMA to transfer the application.

After the code has been transferred to internal memory, a branch instruction is performed. The `IBOOT` signal selects only the upper 2 Kwords of memory as either internal or external, not the entire instruction space. To get to internal memory, jump or branch to an address outside of the 2 Kword boot space and it will force execution from internal memory. Before branching to the new application, file `copy_app` sets the UVT bit in the `%smode` control register to locate the interrupt vector table at 0x0 instead of 0xF800.

`User4.S` is the new main application program which will be loaded into internal memory from flash, then executed from internal memory. It flashes the LEDs with a pattern, with all but one of the lower eight LEDs held

off. The application includes a new interrupt vector table which gets all vectors from ROM. This supports JTAG debugging but not serial-based debugging.

`Boot4 . lcf` contains the linker script commands. There are three relocated sections: the initial boot code (`copy_app`), the new interrupt vector table (part of `User4 . S`) and the new application (`User4 . S`). All three of these sections are linked at one address while being loaded to another address.

The flash programming application requires some program space, which limits the size of the end application to be loaded into flash. One solution to this would be to store part of the new application in data memory and program some section of the flash from data memory instead of instruction memory.

The only way to write internal instruction memory while executing from external instruction memory is to be in the boot area. If the `dir`, `fie`, `lis`, and `sis` bits in the `%smode` control register are set, execution and load/stores take place in external memory. To copy from external to internal instruction memory while executing from external instruction memory, then the code must be executed from the upper boot space, `0xF800–0xFFFF`, and the `IBOOT` pin must be LOW to force external instruction memory. `IBOOT` is used to determine the instruction fetch space, not the load/store space.

[Example 3](#) shows a loop to copy from external instruction memory to internal instruction memory.

Example 3 Loop to Copy External to Internal Instruction Memory

```
! Copy 0x2000 words from 0x0000 in External Instruction Memory to
! 0x0000 in Internal Instruction Memory. After copy, jump to 0x0000
! in Internal Instruction Memory and start executing.

/* Start of code to Manually copy from external IRAM to Internal IRAM */
    mov r7, 0x1fff ! loop counter to 0x2000-1
    mov %loop0, r7 ! initialize loop counter
    mov r0, 0x0000
    mov r1, 1
loop_start:

    mov r4, r0 ! set address
    ! Read word from external instruction memory
    mov r6, %smode
    bits %smode, 5 ! SET_LIS
```

```

bits %smode, 1 ! SET_DIR before loading

ld r5, r4 ! load new word

mov %smode, r6 ! restore smode

! Write word to internal instruction memory

mov r6, %smode

bits %smode, 4 ! set SIS

bitc %smode, 1 ! clear DIR before storing

st r5, r4 ! store word to internal memory

mov %smode, r6 ! restore smode

add r0, r1 ! increment pointer

agn0 loop_start

/* End of manual copy code section */

```

6. Main Program and Link to Files

The files for this project are located on the VeriSilicon website at <http://www.verisilicon.com>.

Example 4 shows the main program for reprogramming the flash memory.

Example 4 Main Program to Reprogram Flash

```

int main(void)
{
    int status;
    unsigned ctl;
    unsigned *ptr;
    unsigned data;
    int erase_stat;

    /* set all accesses to asynchronous */
    ptr = (unsigned *) MMR_MEMMODE;
    data = *ptr;
    *ptr = 0x10;

    /* Set wait states for external accesses to memory and peripherals */
    mov r4, r0! set address
    set_wait_states ();
    /* selects ICSN0 for FLASH Page 0 */

    ctl = P0_ICSN0;
    /* Turn off any buffering in external memory controller while programming flash */ DISABLE
    _STORE_BUFFER

    /* Erase flash, then program the new contents.
    /* The entire flash is erased in this example.
    /* The new application is copied from 0x2000 internal RAM to 0xF800
    /* in external memory for 0x800 words. */
    erase_stat = BRD_FLASH_Erase ( ctl ); /* instruction space - page 0 */
    status = BRD_FLASH_Write_Block((INT16U *)0x2000, (INT16U *)0xF800, 0x800, ctl, INSTRUCTION);
    DEV_MEM_Wr_Mmr (MMR_MEMPCR, 0);

    /* Write to LEDs to indicate status: scrolling is OK, flash is error */
    if (status == WRITE_NO_ERROR)
    {

```

```
/* Write a 0 to 0xffff to solve a interrupt nesting problem with early silicon revs.
/* Debugger does a reti to ffff which is a branch to itself, then next interrupt
/* can take place. Relates more to JTAG debugging than UART debugging */
status = BRD_FLASH_write_word( (INT16U *) 0xffff, 0x0000, ctl);
    for (;;)
    {
        DEV_TIM_Wait (1000);
        scroll_leds ();
    }
}
else
{
    for (;;)
    {
        DEV_TIM_Wait (1000);
        flash_leds ();
    }
}
```