Application Report
SPNA236–March 2017

Sharing Exception Vectors on Hercules™-Based Microcontrollers

Christian Herget

ABSTRACT

Some applications require sharing the exception vectors between different applications, like a bootloader and the main application (firmware). This application report describes a software-based approach to move the exception vector table from the flash into the SRAM, to allow for a flexible assignment scheme and ultimately sharing of the exception vectors.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spna236.

Contents

1 Motivation 1
2 How to Efficiently Move the Exception Vector Table into the RAM 2
3 Additional Considerations 5
4 Example 8
5 References 12

List of Figures

1 Concept With LDR PC, Label 3
2 Concept With Two Tables in Flash and SRAM 4
3 Flow Chart With Two Tables in Flash and SRAM 5

Trademarks

1 Motivation

Customers often ask how the exception vectors can be shared and more importantly the handler routines can be switched between different applications running on the same controller. The background often is, that the customers have up to three applications, a bootloader, the main application (firmware) and a fallback application. Each of the three applications might come with their own exception and interrupt handers and therefore need to share the exception vectors of the ARM® Cortex®-R CPU cores.

However, the handlers for the following four exceptions, the Undefined Instruction Abort, the Super Visor Call, the Prefetch Abort and the Data Abort are usually hardcoded in the exception vector table, which is always located at the start of the flash memory at address 0x0. The so called HIVECS feature of the used ARM Cortex-R CPU cannot be used for this purpose on Hercules-based controllers as these have no memory implemented at the HIVECS address (0xFFFF0000).

To overcome this limitation the approach described on the following pages was developed to allow for efficient relocation of the four mentioned exception vectors into the embedded SRAM, where they can then flexibly be assigned to a specific handler routine by each of the different applications.

2 How to Efficiently Move the Exception Vector Table into the RAM

The exception vector table consists out of eight entries that are usually simple branch instructions to the exception handlers. For more details about the exception handling, see Interrupt and Exception Handling on Hercules™ ARM® Cortex®-R4/5-Based Microcontrollers.

A typical exception vector table (@ address 0x0) on Hercules ARM Cortex-R based MCU's look similar to Example 1.

Example 1. Typical Exception Vector Table

```
        ; interrupt vectors
        resetEntry:
0x00        b   _c_int00
        undefEntry:
0x04        b   undefEntry
        svcEntry:
0x08        b   svcEntry
        prefetchEntry:
0x0C        b   prefetchEntry
0x10        b   _dabort
0x14        b   phantomInterrupt
0x18        ldr pc,[pc,#-0x1b0]
0x1C        ldr pc,[pc,#-0x1b0]
```

The first five entries are direct branches to the exception handlers or themselves, in the case they are not implemented (endless loop). The sixth entry is reserved on Cortex-R4/5 CPU's and usually contains a branch to a handler for so called phantom interrupts. The last two entries are for the Interrupt Request (IRQ) and Fast Interrupt Request (FIQ) and are usually implemented as load instructions with the Program Counter (PC) register as target.

2.1 Some Notes About the Reset, IRQ and FIQ Vectors

As you can see from the vector table in Example 1, the first entry (reset vector) is implemented as a simple branch and the last two entries for the IRQ and FIQ are implemented as load instructions with the PC register as target.

The first entry, the reset vector, has to always point to the main hardware initialization routine, typically to the one from the bootloader, as it has to be executed right after any reset. Note that a CPU reset might need special handling in the bootloader as this is usually triggered within the LBIST sequence.

The last two entries are used to load the interrupt vector address from the Vectored Interrupt Manager (VIM) into the PC register. The load addresses are also relative to the PC to be able to efficiently address the right registers in the VIM memory space. Fore more details, see Interrupt and Exception Handling on Hercules™ ARM® Cortex®-R4/5-Based Microcontrollers.

Because, that the vectors for the IRQ and FIQ interrupt handlers are usually dispatched by the VIM, it is not necessary to relocate these vectors to the RAM as the VIM already offers grate flexibility on remapping and prioritizing interrupts (IRQ and FIQ).

2.2 First Approach (Direct Branch)

One approach to switch between different handlers could be, to replace the entries with a branch instruction to a location in the SRAM with a similar table which then could easily be reconfigured during runtime.

A simple branch instruction in the ARM instruction set takes a 24-bit value to perform a PC relative jump to a specific address (+-32MB). A 24-bit value is not large enough to bridge the gap between the vector tables at address 0x00000000 in the flash to the start of the SRAM at address 0x08000000. By trying to insert a branch which exceeds the 24-bits the Linker will automatically insert a so called trampoline function to perform the jump. Also this might work, using a trampoline function will not be very efficient as it will add extra cycles and will be needed for each of the entries so potentially up to eight times.

2.3 Second Approach (PC relative load)

To overcome the 24-bit (+-32MB) limitation of a branch instruction a load instruction with the PC as the target register can be used. This is similar to what is shown in Example 2 for the IRQ and FIQ. However, this approach requires a small additional (vector) table storing the vectors of the exception handlers in the SRAM memory. Figure 1 shows how this concept works.

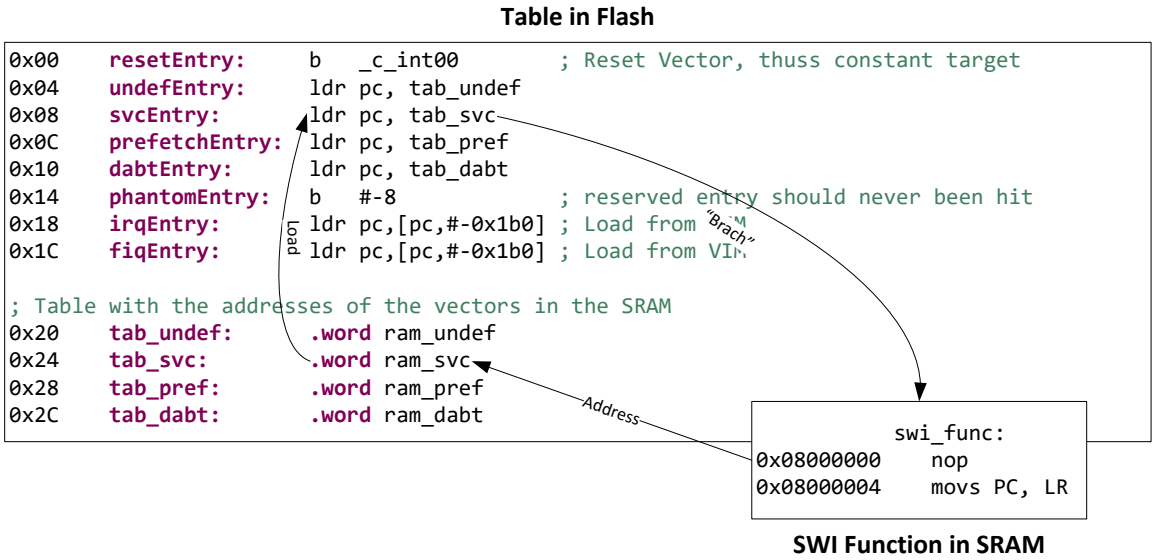

Figure 1. Concept With LDR PC, Label

The method with the load (ldr) instruction discussed in this section has one big advantage compared to the one with the direct branches discussed in Section 2.2, as it keeps the linker from generating additional code for the so called trampolines and therefore is faster and slimmer.

With this approach the exception handlers can already be placed in the RAM, but the start addresses of the routines have to be on fixed known places in the SRAM as the table in the flash can't be changed during runtime. This has the disadvantage that the largest size for each of the exception handlers of the different application images has to be known or that the size of the handlers has to be restricted, both requirements aren't very practical. To overcome this issue the same table based approach can be used to place yet another table in the SRAM. The table in the SRAM can then be dynamically adjusted to the applications needs.

Figure 2 shows the enhanced concept:

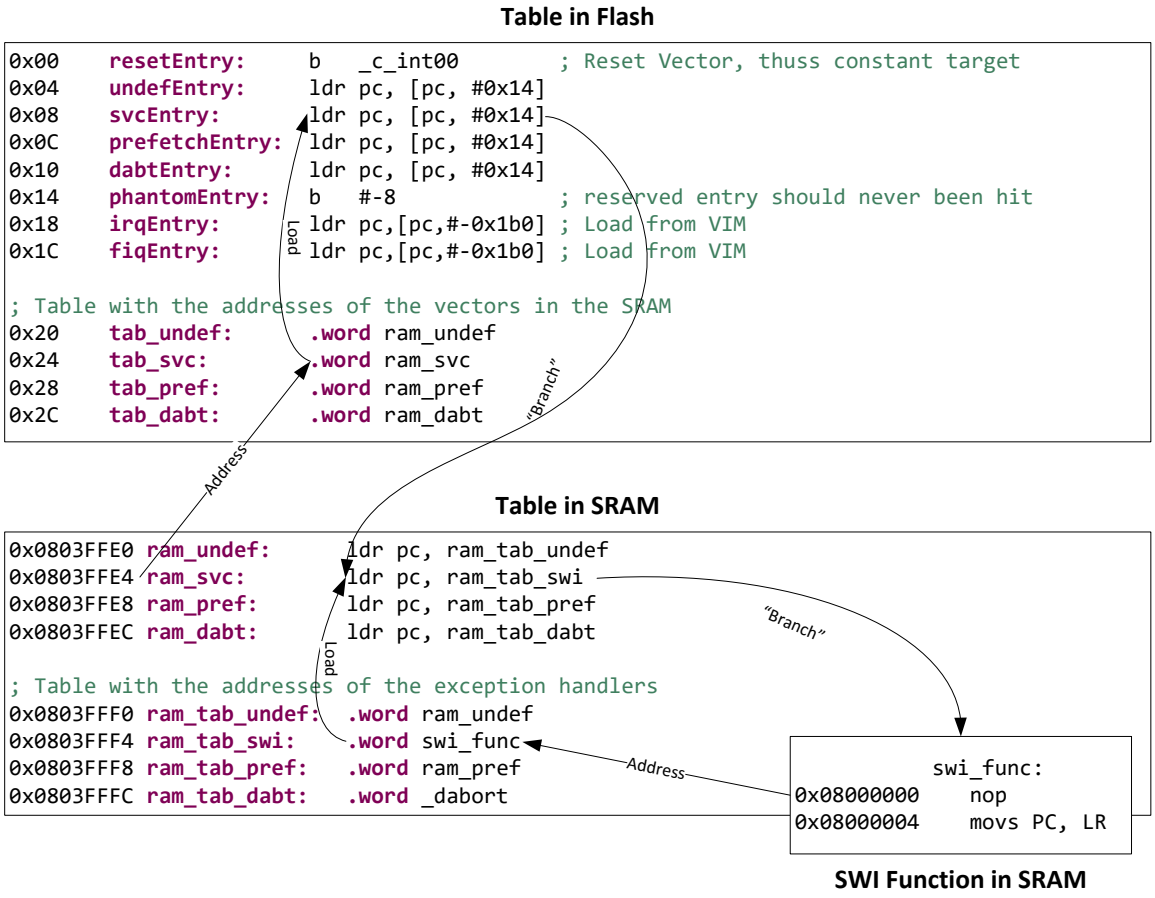

Figure 2. Concept With Two Tables in Flash and SRAM

This concept can now easily be adapted to the applications needs as the actual exception handlers can be placed in either the Flash or SRAM and flexibly assigned by the application by simply altering the table in the SRAM. Figure 3 shows another way to look at this flow, for the example of a Software Interrupt (SWI) (sometimes called Super Visor Call (SVC)).

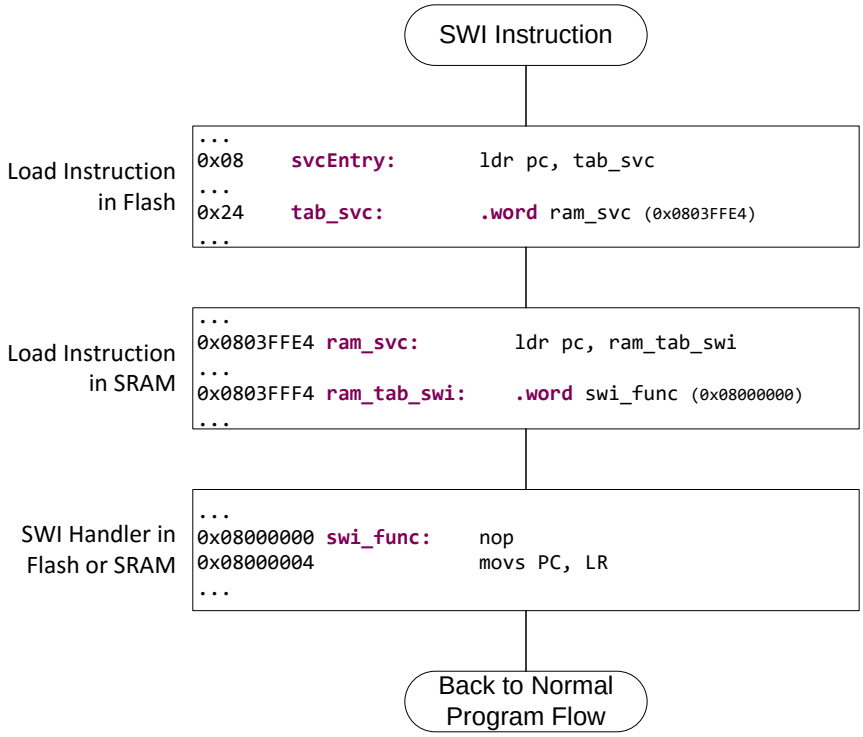

Figure 3. Flow Chart With Two Tables in Flash and SRAM

2.4 Third Approach (Mem Swap)

The Hercules architecture offers the possibility to swap the flash and the SRAM memory locations. With this memory swap, the SRAM starts at address 0x0 where the exception vectors are placed. However, this approach alters the default memory map and comes with its own set of issues that are not discussed further at this point.

3 Additional Considerations

Section 1 discusses how the exception vectors can efficiently be moved to another memory location, for example, into the SRAM. The method with the PC relative load instructions discussed in Section 2.3 were proposed as the preferable approach as it involves the least amount of code and yet is very efficient.

However, there is more to consider then just adding this table, especially when the exception handlers are located in the SRAM. The following sections describe some of these considerations; more might need to be considered on an application level.

3.1 Uninitialized Exception Vector Table

3.1.1 SRAM Content is Unpredictable After a Power On Reset (POR)

It has to be mentioned that the SRAM content is unpredictable after a power on reset. Therefore, it is unpredictable what happens if the CPU takes an exception before the SRAM was initialized with the right content.

Note that an unexpected abort, in general, and before the SRAM got initialized usually indicates a severe issue such as a corrupted memory cell. Expected aborts happen as part of the (initial) self-test routines. Therefore, it can be said that it is unlikely that an abort happens at all during this short time period (reset to start of the software driven self-test routines that caused the aborts).

3.1.1.1 Different Types of Aborts

An undefined instruction abort indicates that the CPU tried to execute an undefined instruction. A prefetch abort indicates an issue while the CPU tried to fetch instructions, which can be caused by corrupted memory cells. A data abort indicates issues with the data load or store unit in the CPU and also can be caused by corrupted memory cells. It has to be mentioned that all these aborts can also be caused by systematic faults in the code.

3.1.1.2 Where to Place the Vector Table in the SRAM

If the exception vector table in the SRAM is placed at the top of the SRAM (high addresses), the CPU executes only a few of these unpredictable instructions before it tries to fetch from an unimplemented memory location, which causes a prefetch abort ideally resulting in an endless loop of prefetch aborts.

This also has the advantage in that the stack typically grows towards the low addresses. Placing the table at the high addresses and the stack at the low addresses therefore ensures that a stack overrun can not accidently overwrite the vector table in the SRAM.

3.1.2 PBIST

The Hercules MCU can test the SRAM cells with an integrated Built-In Self-Test (BIST) module. The SRAM content changes while the test is running and also depends on the selected RAM test algorithm. Therefore, the content can be threaded as unpredictable while the BIST is running. For more details, see Section 3.1.1. The content after the test execution depends on the selected test. Running the recommended MARCH13N on a RM42 results in all 0xC3, the value 0xC3C3C3C3 represents the following ARM instruction:

```
BICGT   R12  R3  #0xC000003
```

This is a data processing instruction and, therefore, does not cause branches to other places in the memory.

NOTE: Check the data left in the SRAM after executing the test algorithm of your choice and on the device of your choice.

3.1.3 TCRAM Auto-Initialization

The Hercules MCU can initialize the SRAM cells to all zero by using the dedicated auto-initialization hardware. This process has to be triggered by software and is usually done early during the boot process.

If an abort occurs, the CPU tries to execute the value all zero in the exception vector table that is in the SRAM. The value all zero represents the following instruction:

```
ANDEQ   R0  R0,  R0
```

This is a conditionally (equal) executed AND instruction without the S suffix (condition code flags are not updated), where the target and source registers are all the same; this instruction can also be interpreted as a No Operation (NOP). If the exception vector table in the SRAM is placed at the top of the SRAM (high addresses), the CPU executes a few of these instructions before it tries to fetch from an unimplemented memory location, which causes a prefetch abort resulting in an endless loop of prefetch aborts.

3.1.4 Adding a Magic Number or Checksum

The proposed mechanisms could be made more robust by adding a magic number or checksum to the table in the SRAM. With this, it would be possible to detect if the table was already loaded with valid content. Note that doing so adds more code and increases the complexity and execution time; therefore, delay association with calling the abort handlers.

3.2 When to Load the Vector Table to the SRAM?

To mitigate the effect of unpredictable SRAM content after a power on reset (see Section 3.1.1), the table should be loaded as early as possible into the SRAM, but only after the SRAM has been (optionally) tested (PBIST), (see Section 3.1.2) and initialized by the TCRAM auto-initialization feature (see Section 3.1.3) and the ECC for the BTCM interfaces are enabled in the CPU.

In the HALCoGen (v04.05.02) supplied startup routine, User Code Section 39 is a good place to add the function call to initialize the exception vector table in the SRAM this is shown in Example 2.

Example 2. Vector Table Copy

```
/* Initialize CPU RAM.
     * This function uses the system module's hardware for auto-initialization of memories and their
     * associated protection schemes. The CPU RAM is initialized by setting bit 0 of the MSIENA
register.
     * Hence the value 0x1 passed to the function.
     * This function will initialize the entire CPU RAM and the corresponding ECC locations.
     */
    memoryInit(0x1U);

/* USER CODE BEGIN (38) */
/* USER CODE END */

    /* Enable ECC checking for TCRAM accesses.
     * This function enables the CPU's ECC logic for accesses to B0TCM and B1TCM.
     */
    _coreEnableRamEcc_();

/* USER CODE BEGIN (39) */
    copy_in(&ramIntvecsCpyTbl);
/* USER CODE END */
```

3.3 How to Protect the Table in the SRAM?

The SRAM content itself is monitored for bit flips by the ECC functionality. However, the ECC functionality does not protect against unintentional overwriting the table. To mitigate this risk, it is possible to program the MPU inside the CPU to block write accesses to the memory region that the vector table is in. Another method would be to periodically read back the content of the table and compare it against the copy in the flash or to periodically calculate and compare a checksum.

The ECC mechanism is not able to correct multi bit errors and might not detect more than the two bit flips in a word. As exceptions like the data abort, prefetch abort undefined instruction abort are expected to occur very rarely, multi bit errors might build up overtime, if not tested (forced, fault injection) periodically. Periodic read back of the entire table can also help to prevent multi bit errors to build up overtime, as the ECC is checked on each read and corrected data word is written back, if needed.

4 Example

The example supplied with this application report implements the concept as discussed in Section 2.3.

There are two example projects: one for RM48L952 and one for TMS570LS3137. The projects have been tested using CCS 7.0.0 with TI ARM compiler v16.9.2.LTS and v5.2.9 using two standard development boards (TMDSRM48HDK and TMDS570LS31HDK) both equipped with the respective silicon revision C.

Both projects are sharing three major source files for this example:

- Linker Command Files
  - RM48_LS31_linker_command_file.cmd → Shared, see Section 4.2
- Support Functions
  - intvecs.asm → Shared, see Section 4.1
  - main.c → Shared, see Section 4.1

The folders for the CCS projects are as following:

- LS31_Exception_Table_in_RAM → CCS Project for TMS570LS3137
- RM48_Exception_Table_in_RAM → CCS Project for RM48L952

4.1 Source Code

The file *intvecs.asm* contains the assembly language code needed to implement the vector forwarding (vector tables). The default section *.intvecs* containing the interrupt vectors at address 0x0 is linked into the Flash memory and the *ramIntvecs* section containing the adjustable vectors into the SRAM.

Example 3 shows the code for the *.intvecs* section. The first entry in the reset vector is a branch directly to the *_c_int00* function that is used to initialize the controller. The following four entries are forwarded into the SRAM: Undefined Instruction Abort, Software Interrupt (or Super Visor Call), Prefetch Abort and Data Abort. The actual addresses are defined in the four entry table shown at the bottom of Example 3. The sixth entry is an endless loop (phantomEntry) this should never be taken by the CPU as it is a reserved entry. The last two entries are the entries for the IRQ and FIQ that are dispatched by the VIM, therefore, these two do not need to be forwarded in software.

Example 3. Code for Section .intvecs

```
;-------------------------------------------------------------------------------
; interrupt vectors in Flash
    .sect   ".intvecs"
    .retain ".intvecs"
    .arm

resetEntry:     b   _c_int00        ; Reset Vector, thuss constant target
undefEntry:     ldr pc, tab_undef
swiEntry:       ldr pc, tab_swi
prefetchEntry:  ldr pc, tab_pref
dabtEntry:      ldr pc, tab_dabt
phantomEntry:   b   phantomEntry    ; Endless Loop, reserved entry should never been hit
irqEntry:       ldr pc,[pc,#-0x1b0] ; Load from VIM
fiqEntry:       ldr pc,[pc,#-0x1b0] ; Load from VIM

;-------------------------------------------------------------------------------
; Table with the addresses of the vectors in the SRAM
tab_undef:      .word ram_undef
tab_swi:        .word ram_swi
tab_pref:       .word ram_pref
tab_dabt:       .word ram_dabt
;-------------------------------------------------------------------------------
```

Example 4 shows the code for the section *ramIntvecs*. It contains the entries for the three aborts and the software interrupt as discussed in the previous paragraph. The entries for the Undefined Instruction Abort and the Prefetch Abort are endless loops in the example. The entries for the software interrupt and the data abort are pointing to the associated handlers. For the data abort handler, the handler supplied with HALCoGen is used, it is placed in the Flash memory. The software interrupt handler is only a stub doing nothing it is placed in the SRAM for demonstration purposes only.

Example 4. Code for Section ramIntvecs

```
;-------------------------------------------------------------------------------
; Table in RAM
    .sect   "ramIntvecs"
    .retain "ramIntvecs"
    .arm

ram_undef:      ldr pc, ram_tab_undef
ram_swi:        ldr pc, ram_tab_swi
ram_pref:       ldr pc, ram_tab_pref
ram_dabt:       ldr pc, ram_tab_dabt

;-------------------------------------------------------------------------------
; Table with the addresses of the exception handlers
ram_tab_undef:  .word ram_undef
ram_tab_swi:    .word swi_func1
ram_tab_pref:   .word ram_pref
ram_tab_dabt:   .word _dabort
;-------------------------------------------------------------------------------
```

Example 5 shows the code for *main()* and two software interrupt handlers called *swi_handler_ram* and *swi_handler_flash* in this example. The main function simply calls *swi_handler_ram* and *swi_handler_flash* and, thus, triggers a software interrupt in the CPU. The attribute keyword is used to tell the compiler that *swi_handler_ram* and *swi_handler_flash* are of the interrupt type SWI and that *swi_handler_ram* should be placed in the SRAM (*ramfunc*) rather than in the flash (.text).

Example 5. Code From main.c

```
#include "hal_stdtypes.h"

/* Define two handlers for the Software Interrupt (Super Visor Call) */
__attribute__((interrupt( "SWI" ), ramfunc )) void swi_handler_ram( void );
__attribute__((interrupt( "SWI" ) ))          void swi_handler_flash( void );

extern volatile uint32 ram_tab_swi;

int main(void)
{
    /* Call swi_handler_flash() from Flash */
    _call_swi(0);

    /* Change the address of the SWI handler */
    ramTabChangeEntry(ENTRY_SWI, &swi_handler_ram);

    /* Call swi_handler_ram() from SRAM */
    _call_swi(0);

    return 0;
}

void swi_handler_ram(void)
{
    return;
}

void swi_handler_flash(void)
{
    return;
}
```

The data abort is triggered during the startup of the device by *Flash ECC Check* self-test supplied with HALCoGen. The software interrupt handlers are triggered in the main function as shown in Example 5, *swi_handler_flash()* is linked into the flash and *swi_handler_sram()* is linked into the SRAM memory. The vector for the software interrupt gets updated (switched) in *main()* between the two SWI calls (*_call_swi(0)*).

This example shows how two exception handlers can be placed in the flash and SRAM and how to dynamically switch between the two handlers. For this to work, it is necessary to copy the vector table and the function *swi_handler_sram()* into the SRAM. The vector table and initial vector table are copied during the initialization of the microcontroller after the SRAM was initialized to all zero and the ECC was enabled, as shown in Example 2. The function *ramTabChangeEntry()* is used to modify the vector table and switch between two handlers.

The code in the project was updated compared to what is shown in Example 2. It now triggers the three aborts (Undefined Instruction, Data, Prefetch) too, in order to be able to test the method of forwarding these exceptions. These changes are not documented here in order to keep the complexity of this documentation low.

4.2 Linker Command File

The linker command file has to be changed compared to the default file supplied with HALCoGen in order to make this example work. Therefore, a new linker command file called RM48_LS31_linker_command_file.cmd was created.

4.2.1 Memory Map

The size of the *.intvecs* section has to be increased to 64 byte in order to fit the code shown in Example 3. Theoretically, 48 bytes would be enough to fit the *.intvecs* section. However, experiments have shown that the current linker versions do have issues with calculating the ECC bits if the memories are not aligned to 32 bytes. A range for the vector table in the SRAM should be added at the end of the SRAM memory (high addresses, *RAMVECTORS*).

Example 6 shows the modified memory map. There are more changes and additions than described in the previous paragraph, mainly the addition of memory ranges for the ECC bits.

Example 6. Memory Map

```
/*----------------------------------------------------------------------------*/
/* Memory Map                                                                 */

#define RAM_SIZE         (256 * 1024) /* 256kB */
#define RAM_STACK_SIZE   (0x1500) /* 5.25kB */
#define RAM_INT_VEC_SIZE (0x20) /* 32Byte*/

MEMORY
{
/* Flash Memory */
/* Bank 0 */
    VECTORS  (X)  : origin=0x00000000                              length=0x00000040
  fill=0xffffffff /* 64Byte */
    FLASH0   (RX) : origin=(end(VECTORS))                          length=(0x0180000 -
 size(VECTORS))    vfill=0xffffffff /* 1.5MB */
/* Bank 1 */
    FLASH1   (RX) : origin=0x00180000                              length=0x00180000
 vfill=0xffffffff /* 1.5MB */
/* Bank 7 (FEE) */
    FLASH7   (R)  : origin=0xF0200000                              length=0x00010000
 vfill=0xffffffff /* 64kB */
/* Bank 0 ECC */
    ECC_VEC  (R)  : origin=(0xf0400000 + (start(VECTORS) >> 3))  length=(size(VECTORS) >> 3)
 ECC={algorithm=algoR4F021, input_range=VECTORS, fill=false}
    ECC_FLA0 (R)  : origin=(0xf0400000 + (start(FLASH0)  >> 3))  length=(size(FLASH0)  >> 3)
 ECC={algorithm=algoR4F021, input_range=FLASH0,  fill=false}
/* Bank 1 ECC */
    ECC_FLA1 (R)  : origin=(0xf0400000 + (start(FLASH1)  >> 3))  length=(size(FLASH1)  >> 3)
 ECC={algorithm=algoR4F021, input_range=FLASH1 }
/* Bank 7 ECC */
    ECC_FLA7 (R)  : origin=0xF0100000                              length=(size(FLASH7)  >> 3)
 ECC={algorithm=algoR4F021, input_range=FLASH7 }
/* embedded SRAM */
    STACKS     (RW)  : origin=0x08000000                           length=(RAM_STACK_SIZE)
    RAM        (RWX) : origin=(end(STACKS))                        length=(RAM_SIZE - RAM_STACK_SIZE -
 RAM_INT_VEC_SIZE) /* 256kB - xyz */
    RAMVECTORS (RWX) : origin=(end(RAM))                           length=(RAM_INT_VEC_SIZE)
}
```

4.2.2 Sections Configuration

Several sections have to be added to the *SECTIONS* configuration in the linker command file. This is mainly necessary because of the use of Linker Generated Copy tables and RAM functions. For more information on copy tables and RAM functions, see the ARM Assembly Language Tools User's Guide. These sections are *.ovly* and *.binit* for the copy tables and *.TI.ramfunc* for the RAM functions. To place the vector table in the SRAM another section has to be added (*ramIntvecs*).

Example 7 shows the modified sections configuration.

Example 7. Sections Configuration

```
/*----------------------------------------------------------------------------*/
/* Section Configuration                                                      */

SECTIONS
{
    /* Flash based sections */
    .intvecs    : {} palign=16, fill=0xff, > VECTORS
    .text       : {} palign=8              > FLASH0 | FLASH1 /* Executable code and constants */
    .const      : {} palign=8              > FLASH0 | FLASH1 /* Global and static const variables
that are explicitly initialized */
    .binit      : {} palign=8              > FLASH0 | FLASH1 /* Boot time copy tables */
    .ovly       : {} palign=8              > FLASH0 | FLASH1 /* User defined Copy Tables */
    .cinit      : {} palign=8              > FLASH0 | FLASH1 /* Tables for explicitly initialized
global and static variables */
#ifdef __TI_EABI_SUPPORT__
    .init_array : {} palign=8              > FLASH0 | FLASH1 /* C++ global constructor addresses */
#else /* TI_ARM9_ABI and TIABI */
    .pinit      : {} palign=8              > FLASH0 | FLASH1 /* C++ global constructor addresses */
#endif

    /* Special TI Sections */
    .TI.ramfunc : {} palign=8, load=FLASH0, run=RAM, table(BINIT) /* Section for the ramfunc
attribute */
    .TI.crctab  : {} palign=8              > FLASH0 | FLASH1 /* The CRC tables generated by the
linker */

    /* RAM based sections */
    .stack      : {__STACK_START = .;} type=NOINIT > STACKS /* Stack */
    .bss        : {} palign=8                      > RAM    /* Uninitialized global and static
variables */
    .data       : {} palign=8                      > RAM    /* Global and static non-
const variables that are explicitly initialized. */
    .sysmem     : {}                               > RAM    /* Memory pool (heap) for dynamic memory
allocation */

    /* run (HIGH) to place the table at the top of the RAM */
    ramIntvecs  : {} load=FLASH0, run=RAMVECTORS, palign=8, table(ramIntvecsCpyTbl)
}
```

5 References

1. Interrupt and Exception Handling on Hercules™ ARM® Cortex®-R4/5-Based Microcontrollers
2. ARM Assembly Language Tools User's Guide

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265