Engineer-to-Engineer Note EE-359

ANALOG DEVICES **Technical notes on using Analog Devices products and development tools**
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

ADSP-CM40x Boot Time Optimization and Device Initialization

Contributed by Andrew Caldwell and Kritika Shahu *Rev 2 – July 7, 2015*

Introduction

The ADSP-CM40x family of mixed-signal control processors provide on-chip programmable SPI flash memory for code and data storage. The SPI peripheral and the implementation of an instruction cache on ADSP-CM40x processors allow for code execution directly from the on-chip SPI flash device.

The processor, when originally released from reset, executes code from the on-chip boot ROM space. This boot code is responsible for initial processor configuration and for handling each of the processor's supported boot modes[1].

The boot process is capable of vectoring and executing code directly from the on-chip SPI flash memory, or it may load a boot image in the form of a boot stream to the processor's internal SRAM.

The purpose of this EE-Note is to introduce users to techniques that may be used to reduce the initial system bring-up time when using the SPI master boot mode for code execution and show how to implement device initialization software to optimize the system hardware as early as possible before executing the end application.

Example code compatible with *IAR Embedded Workbench for ARM®*[4] development tools is provided in the *ADSP-CM40x Enablement Software Package*[3] .

Optimizing Boot Time

In order to minimize processor bring-up time, it is important to optimize system clocks and configure core features/peripherals as early as possible in the boot process. By default, the processor exits the reset state with the PLL in bypass mode to ensure that the oscillator watchdog - a peripheral used to detect the most probable oscillator failures, such as loss of input clock or harmonic oscillation - can be configured appropriately before bringing the PLL out of bypass mode.

In order to optimize the bring-up time for SPI master boot mode, it is also required to configure the SPI peripheral and cache controller for optimal performance. The SPI flash memory can be configured in command skip mode, and the cache pre-fetch feature can be enabled in order to optimize execution of code directly from internal flash.

There are a number of ways to configure the system with processor initialization routines for the Clock Generation Unit (`CGU`), Dynamic Power Management block (`DPM`), Oscillator Watchdog (`OSCWD`), Serial Peripheral Interface (`SPI`), and cache controller.

1. Initialize all required units from *main()* in the user application code.
2. Use an initialization block, if booting the processor using a compliant boot stream consisting of block headers and payload.
3. Implement a multi-application approach, where device initialization is maintained in a separate piece of firmware from the end application.
4. Utilize the *__low_level_init()* function of the *IAR Embedded Workbench* run-time startup.

This document will focus on the latter two options to configure the processor efficiently and early in the boot process prior to the rest of the runtime initialization sequence. A brief introduction to the other two methods is also provided.

The selected methods are two distinctly different approaches to the same problem, each having its own benefits and limitations. By providing examples of how to implement these two approaches, users can adopt a strategy that is best suited to their requirements.

Initializing the Units from main() in the Final Application

Initializing all units at the beginning of the application code is the least efficient way to configure the processor and results in extended bring-up time.

The `DPM` and `CGU` blocks are just some of the components that may be required to be configured. It is advisable to execute the initialization routine from internal SRAM, as the clock being supplied to the SPI flash device will also be reconfigured during the process.

The run-time setup code for the user application will copy all code and data intended for internal SRAM during the execution of the startup sequence before executing the main routine where the hardware is then optimized. Since the PLL is bypassed during the copying operation, the bring-up time is extended.

For those applications that are not concerned about boot time requirements, this is certainly a viable option and perhaps the simplest to implement. A single project is required, with the only requirement being that code brings the PLL out of bypass and reconfigures the `CGU` block to be executed from SRAM (not from SPI flash memory).

Initialization Codes from Boot Streams

The on-chip boot ROM provides a boot kernel that is fully capable of loading a user application in a distributed manner to the on-chip SRAM, provided that the application image is in a compliant boot stream format. This boot stream format supports a feature referred to as *'init code'*. During the booting process, a block header instructs the boot kernel that init code has been loaded. The boot kernel will then execute this code before continuing with the boot process. It is an effective means of optimizing the system early in the boot process before the rest of the user application has been loaded. For further details, please refer to the *Boot ROM and Booting the Processor* section of the *Hardware Reference Manual*[2].

Using Multiple Applications for Initialization

The multiple application approach requires the user to create and load a small application solely for the purposes of configuring the optimization features of the processor. As the application is small, a minimum amount of user code and data are loaded from the SPI flash memory to the internal SRAM before being executed. Once executed, the application vectors to the next application in the flash memory space, whether that is a second-stage boot loader or the end application. This method minimizes the number of SPI transactions and instructions executed at the slower clock frequency.

For ease of use, each individual application should be loaded to a separate erasable block of memory in the on-chip flash memory. The Interrupt Vector Table (`IVT`) of processor optimization firmware should be placed at the `bootAddress` memory location defined by the security header. When the main user application code is in development or being updated in the flash device, the software responsible for optimizing the processor configuration remains intact in a separate erasable memory block.

Second-stage boot loaders may be implemented such that they are called after the initial application responsible for optimizing the processor configuration and prior to the main user application. These second-stage boot loaders may provide firmware update functionality for all the applications stored in the flash device. This type of approach provides the benefit of the processor optimization firmware and the second-stage loader firmware being completely independent from the end user application. The maintenance of the various application images and update functionality is taken care of solely by the second-stage boot loader.

While the end user application is not required to have any knowledge of the other applications executed previously, the linker files for the main application must not make use of the flash memory banks that are intended to hold the other pieces of firmware. Therefore, maintenance of multiple projects and linker files is required. The use of multiple applications within the flash memory in conjunction with a second-stage boot loader for firmware update functionality provides an extremely flexible solution for devices requiring field upgrade functionality while still supporting quicker bring-up time.

Using __low_level_init() for Initialization

Beside the method of initializing everything from the *main()* routine, this is likely the simplest approach to implement. Only a single project is required, which means that a single linker file is involved. This method does not provide the flexibility of having the initial processor configuration and second-stage boot loader for individual firmware image maintenance, and any update to the software would require the entirety of the flash memory to be reprogrammed.

The *__low_level_init()* routine is called early in the runtime startup sequence, providing a hook to initialize hardware prior to the rest of the runtime initialization in which all the code and data is copied from flash memory to internal SRAM space. This approach may take slightly longer than the multiple application approach, depending on the functionality implemented within the `startup.c` file prior to the call to the *__low_level_init()* routine. For example, if the processor's entire vector table is relocated from flash memory to internal SRAM, this may take place before the *__low_level_init()* function call; thus, it will be happening prior to the clocks being optimal. Users may wish to reorganize the operations performed in the `startup.c` file for their specific requirements.

Executing Code from SPI Flash Memory

Details on the SPI master boot mode can be found in the hardware reference manual[2]. The general procedure in the SPI master boot mode is that upon completing the pre-boot sequence, the processor will proceed to boot from the on-chip SPI flash memory via the SPI2 interface. If no boot stream is found, then the processor will check for a valid stack pointer in the first entry of the vector table. The reset vector of the vector table is located at the next physical address.

If the stack pointer is found to fall within valid SRAM space, then the processor will branch to the address stored in the reset vector. By this time, the SPI peripheral has been configured to allow for code execution directly from the SPI flash memory. At this point, the processor is running in PLL bypass mode and the `CGU` is not configured, hence the SPI code execution speed is not optimal.

In general, an application loaded to the SPI flash memory for execution would have data and code sections that are required to be copied from the SPI flash memory to the internal SRAM space during the run-time initialization phase and prior to executing *main()*. These memory sections to be initialized and copied are all controlled by the project linker file.

The example below highlights the linker command to perform this type of operation. For further details, please refer to the *IAR C/C++ Compiler, Compiling and Linking Manual*[5].

```
initialize by copy    {rw};
```

In order to minimize the boot time, the initialization time of the SRAM memory from the SPI flash content can be improved by optimizing all the system clocks prior to running the C-run-time for the main application. This will also improve the performance of code being executed from flash memory, as the cache line instruction fetches will be more efficient.

Optimization and Initialization Operations

The examples discussed apply the same optimizations and configuration, but use different methods. Each of the examples:

- Configures the Oscillator Watchdog,
- brings the processor out of PLL Bypass Mode,
- optimizes the various system and core clock frequencies,
- configures the SRAM memory partitioning,
- initializes any newly added code or data sections for parity error detection,
- optimizes the SPI interface and configures the flash memory for command skip mode,
- enables posted-write functionality, and
- enables the cache pre-fetch feature.

The example software provided in the *ADSP-CM40x Enablement Software Package*[3] has been developed with minimal error handling support in order to keep the footprint small. The examples can be easily expanded depending upon requirements to provide more robust error handling mechanisms.

Using Multiple Applications for Processor Optimization

The example discussed in this section requires the development and maintenance of two individual projects. The '*proc_init*' project is responsible for configuring and optimizing the processor, and the '*blink*' project in this case is the main user application to be executed.

An *IAR Embedded Workbench* workspace file is located in the following folders of the enablement software package:

- '*Boot_Optimization_Multi_App\blink\CM403F\iar*'
- '*Boot_Optimization_Multi_App\blink\CM408F\iar*'

Figure 1 shows the workspace containing the two projects.

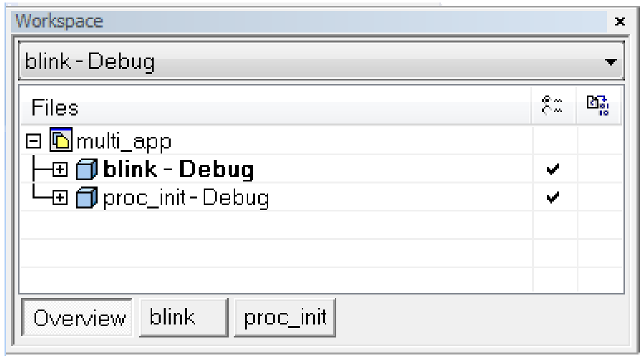

Figure 1. Multi-Application Workspace

The following routines in the '*proc_init*' project are copied to SRAM memory space for execution:

- `main()`
- `init_cgu()`
- `init_dpm()`
- `init_spi()` (and all additional SPI configuration routines)

The following routine is configured to be located in and executed from flash memory space:

- `init_mem()`

The `__ramfunc` keyword is used to instruct the linker to execute code from SRAM space. There are different methods of implementing placement of functions and copying them from flash memory to SRAM space for execution. Pragma directives may be used to place the code in a specific section. That section can then be marked as `initialize by copy` in the linker file.

The `__ramfunc` keyword provides additional information when compiling the code. Diagnostic warnings are provided indicating the possibility that a function declared using the keyword may be accessing data not located in SRAM space. There may be cases where statically initialized local variables result in a constant being located in the flash space. If a function, such as one that is required to reconfigure the SPI flash memory or perform an SPI program operation, executes and requires access to those constants located in the SPI flash memory to initialize the local data, then the code may fail. The SPI peripheral may not be configured for the required memory mapped read mode of operation.

Listing 1 shows the use of the `__ramfunc` keyword.

```
__ramfunc void main()
{
  uint32_t app_address = 0;
  uint32_t dummy = 0;

  /* Bring the PLL out of bypass mode */
  if(init_dpm() != INIT_DPM_RESULT_SUCCESS)
  {
    /* Call the error handler */
   …
   …
  }
```

Listing 1. Using the __ramfunc Keyword

The critical parts of the linker file showing the memory placement and the various section initialization requirements are shown in Listing 2.

```
define symbol FLASH_SIZE                    = 0x00001000; // 4 KB Erasable sector
define symbol FLASH_BASE                    = 0x18001000; // Memory mapped base address

define region FLASH_region                  = mem:[from FLASH_BASE size FLASH_SIZE];

// INITIALIZATIONS...
do not initialize                           { section .noinit };
do not initialize                           { section .mainstackarea };
do not initialize                           { section .processstackarea };

initialize by copy                          { rw };

// flash code
place in FLASH_region                       { ro };

// SRAM Code (regular .text plus .textrw from "__ramfunc")
place in RAM_code_region                    { rw section .text, rw section .textrw};

// SRAM data
place at end of RAM_data_region             { block CSTACK };
place in RAM_data_region                    { rw, block HEAP };
```

Listing 2. Using the __ramfunc Keyword

For this example, the application code and data must be mapped to the second erasable 4 KB sector in flash memory. The security header is placed at the beginning of the flash at location `0x18000000`. The security header contains a `bootAddress` field that defines the default address to boot from or the location of the interrupt vector table when code is to be executed from flash. In this case, the `bootAddress` field is `0x18001000`, hence the interrupt vector table for the '*proc_init*' project is placed at that location, and the code and data are then mapped thereafter.

A large portion of the implementation of the default `startup.c` file that is supplied with the examples included in the enablement software package has been removed in order to minimize the amount of software to be copied and executed during this phase of the boot process.

The interrupt vector table for the boot ROM code has been left as the active vector table. The *proc_init* vector table only contains two entries, the stack pointer and the reset vector, which is the minimum required in order for the boot code to successfully transition to the application stored in flash memory.

Once the hardware initialization has completed, the routine must vector to the next application. For this example, the next application has been mapped to SPI memory address `0x00010000`, which correlates to the memory-mapped address `0x18010000` and is a non-occupied erasable block of flash memory.

An example function responsible for transitioning to the next application is shown in Listing 3. This function takes a pointer to the application's vector table, reads the stack pointer from the vector table, and sets the core main stack pointer before reading the reset vector and branching. Additional error checking may be implemented in order to check that the vector table is valid.

```
inline void call_app(uint32_t address)
{
  __ASM("ldr r1, [r0, #0]");
  __ASM("msr msp, r1");
  __ASM("ldr r1, [r0, #4]");
  __ASM volatile ("bx      r1");
}
```

Listing 3. Example Transition Code between Applications

The application to be executed after the processor initialization sequence is a simple blink program. In order to bulk up the application, some data sections are also included. This is simply intended to better highlight the advantages in the boot time when the application requires more data to be copied from SPI flash memory to SRAM internal memory space than the simple blink application provides.

The linker file for the blink program is required to be modified so that the application does not get mapped to the same flash sector as the hardware initialization section, as shown in Listing 4.

```
// 2048k FLASH
define symbol FLASH_SIZE  = 0x001F0000; // 2M byte Flash, reduced by base offset
define symbol FLASH_BASE  = 0x18010000; // Internal stacked QSPI (SPI2) Flash
```

Listing 4. Linker File Snippet for Blink Project

The blink routine is placed at `0x18010000` in the flash, which is where it will then be executed from. The linker file is modified to offset the base location of flash memory in order to take into consideration the hardware initialization firmware located in the first block.

Using __low_lev_init() for Processor Optimization

IAR Embedded Workbench run-time provides support for early initialization of hardware features prior to the rest of the run-time initialization, where a bulk of the data would be copied from flash memory to SRAM space. In order to make use of this feature, users can add a *__low_level_init()* function to their application, which can be used to configure the hardware.

First, the application software that is required to execute from this function must be considered. Statically initialized variables that may need to be copied from flash memory to SRAM space cannot be used, as the routine is executed prior to the software that performs all the variable initialization. As previously described, a bulk of the software executed is to be copied from the flash memory to SRAM space and then executed. Once again, the routines that perform this task of copying the code from flash memory to SRAM space are not called until afterwards.

For hardware initialization tasks that simply need to execute from flash memory early in the boot cycle, this is the ideal location for them. In order to execute code from SRAM space, however, some additional linker functionality is required. The linker in the previous multi-application case used the `initialize by copy` directive for instructing the linker to arrange the content for initialization by copying the section from flash memory to SRAM space. The linker also has an `initialize manually` directive. Sections declared with this directive will not be copied during the run-time process by the standard run-time software and must instead be copied manually with additional code.

All routines to be executed from SRAM space are not only defined using the `__ramfunc` keyword, but they are also explicitly placed within a section using the `location` pragma, as shown in Listing 5.

```
#pragma location = "hw_init"
__ramfunc void main()
{
  uint32_t app_address = 0;
  uint32_t dummy = 0;

  /* Bring the PLL out of bypass mode */
  if(init_dpm() != INIT_DPM_RESULT_SUCCESS)
  {
    /* Call the error handler */
  …
}
```

Listing 5. Using location Pragma to Place Routines

An example implementation of the *__low_level_init()* function is shown in Listing 6. The implementation shows how all the data from flash memory can be manually copied to SRAM space in order to execute the code, all prior to the rest of the runtime initialization. The function is preceded with two pragma commands, which allow for dedicated section operators to be used to determine the source and destination locations and sizes of the sections to copy.

```
#pragma section = "hw_init"
#pragma section = "hw_init_init"
uint32_t __low_level_init()
{
  char * from = __section_begin("hw_init_init");
  char * to = __section_begin("hw_init");

  for(uint32_t i = 0; i<__section_size("hw_init"); i++)
    to[i] = from[i];

  proc_init();
  return 1;
}
```

Listing 6. Example __low_level_init() Function

In addition to the code required to copy the data, adjustments to the linker file are also needed to perform this task, as depicted in Listing 7.

```
// ICF FILE INITIALIZATIONS...
do not initialize                    { section .noinit };
do not initialize                    { section .mainstackarea };
do not initialize                    { section .processstackarea };
initialize manually                  { section hw_init };

// SRAM Code (regular .text plus .textrw from "__ramfunc" and hw_init)
place in RAM_code_region  { rw section .text, rw section .textrw,
                            rw section hw_init};
```

Listing 7. Linker File Commands for Manual Initialization

Two sections are required when `initialize manually` is used. The section name containing the data to be copied is appended with "`_init`", and the section in which the data is to be copied to is the same as the section name used. In the previous example, if `initialize manually` were used for a section named "`hw_init`", the data to be copied would be located in a section named "`hw_init_init`", and the section it would be copied to would be named "`hw_init`".

Conclusion

The two methods of initializing the system hardware early in the boot phase can significantly reduce the bring-up time of the processor. In order to highlight the benefits, a simple benchmark was taken in which the blink application initialized the system hardware after entry to the *main()* routine. The application was run on an ADSP-CM408F EZ-KIT® evaluation platform. The software for each of the examples was compiled using the same compilation settings.

The time measured was from the de-assertion of the `SYS_RESOUTb` signal to the first toggling edge of the GPIO (toggled at the beginning of the main) on the ADSP-CM408F EZ-KIT platform. The boot ROM code de-asserts the `SYS_RESOUTb` signal once the initial pre-boot phase has completed and prior to executing the actual boot mode code.

Table 1 below shows the results when initializing the system hardware from *main()* as opposed to using the multi-application and *__low_level_init()* implementations for CCLK of 240 MHz and SCLK of 96MHz.

Hardware Initialization Method	Bring up time
Initialize from main()	21.11 ms
Multi Application	1.277 ms
__low_level_init()	1.594 ms

Table 1. Boot Time Benchmarks

The most efficient method was to use multiple applications, but this was very closely followed by the *__low_level_init()* implementation. The *__low_level_init()* implementation was slightly slower, as the initial startup for that application copied the entire interrupt vector table from flash memory to SRAM space prior to optimizing the system hardware. The system startup could be rearranged to further optimize this process, however the benefits would be minimal. The multi-application implementation initially only copied two entries of the vector table before then optimizing the hardware. The benefits of implementing such techniques become more significant as the application requirements grow and the amount of data to be copied from flash memory to SRAM space increases.

Developers who are looking to implement *In Application Programming (IAP)* that can be stored in the initial boot sector of the flash may wish to adopt the *__low_level_init()* functionality for the *IAP* to minimize the number of firmware images that are required to be maintained, leaving only the *IAP* and the end application. The *IAP* will always be executed first, optimizing the hardware configuration before then deciding whether a firmware update is being requested. If no firmware update is required, the *IAP* can then use the techniques highlighted in the multi-application method to vector to the main application.

Firmware update software is often required to be copied to internal SRAM memory before being executed, allowing for dynamic updates to flash content. The techniques highlighted for copying and initializing regions of memory both prior to and during the runtime allow for *IAPs* to be executed initially and with low impact to the main application's bring-up time (in the event a firmware update is not required).

References

[1] *ADSP-CM40x ARM Cortex-M4 Mixed-Signal Control Processor Data Sheet.* Rev PrG, May 2015. Analog Devices, Inc.

[2] *ADSP-CM40x Mixed-Signal Control Processor Hardware Reference Manual.* Preliminary Rev 0.2, September 2013. Analog Devices, Inc.

[3] *ADSP-CM40x Enablement Software Package.* Rev 2.1.0. Analog Devices, Inc.

[4] *IAR Embedded Workbench for ARM (http://www.iar.com/). 7.40.2.* IAR Systems AB.

[5] *IAR C/C++ Compiler, Compiling and Linking Manual (http://www.iar.com/). 9th Edition May 20012,* IAR Systems AB.

Document History

Revision	**Description**
Rev 2 – July 7, 2015 *by Kritika Shahu*	Revised for Silicon Rev H
Rev 1 – September 5, 2013 *by A. Caldwell*	Initial release.