ADSP-CM40x Boot Time Optimization and Device Initialization

Contributed by Andrew Caldwell

Rev 1 – September 5, 2013

Introduction

The ADSP-CM40x family of mixed-signal control processors provides 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 processors 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 processors 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 how to implement device initialization software to optimize the system hardware as early as possible before executing the main user application.

Example code compatible with IAR Embedded Workbench for ARM® 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, configure core features and peripherals as early as possible in the boot process. The processor, by default, exits the reset state with the PLL in bypass mode. This is 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.

There are a number of means that may be used in order to configure the Clock Generation Unit (CGU), Dynamic Power Management block (DPM) and the Oscillator Watchdog (OSCWD).

1. Initialize all required units from main() in the user application code.
2. Use Initialization codes if booting the processor using a compliant boot stream consisting of block headers and payload.
3. Implement a multi-application based approach in which device initialization is maintained in a separate piece of firmware from the end application
4. By utilizing the __low_level_init() function of the IAR Embedded Workbench run-time startup.
This document will focus on the latter two implementations of using multiple applications to maintain a separate device initialization firmware image and the use of the \_low\_level\_init() method of the runtime in order 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 with 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 users 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 configure the modules by executing the software 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.

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 reconfigure the CGU block to be executed from SRAM and not from SPI flash memory space.

**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 if the application image is converted to a compliant boot stream format. This boot stream format supports a feature referred to as ‘init codes’. During the booting process a block header instructs the boot kernel that an 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\[2\].

**Multiple Applications**

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 space, whether that is a second stage boot loader or the end user 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. 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 on 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 for the other pieces of firmware. Therefore, maintenance of multiple of projects and linker files is required. However 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 in field upgrade functionality while still supporting quicker bring-up time.

**Using __low_level_init()**

This is likely the simplest approach to implement excluding the method of initializing everything from the main() routine. Only a single project is required, thus just a single linker file. This method does not provide the flexibility of having the initial processor configuration and second stage boot loader for individual firmware image maintenance. Any update to the software requires the reprogramming of the flash for all components.

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 into 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 function call, and 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**

Details on the SPI master boot mode can be found in the ADSP-CM40x mixed-signal control processor 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 located at SPI physical address 0x00000000. The reset vector of the vector table is located at physical address 0x00000004.

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. The SPI code execution speed is not optimal as the CGU is not configured.
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 using different methods. The following operations are performed:

- Oscillator Watchdog configuration
- Bringing the processor out of PLL Bypass Mode
- Optimizing the various system and core clock frequencies
- Configuring the SRAM memory partitioning
- Initializing any newly added code or data sections for parity error detection
- Optimizing the SPI interface
- Enabling posted-write functionality

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`

The workspace contains the two projects.
The following routines are copied to SRAM memory space for execution:

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

The following routines are configured to be located 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. Pragmas 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.

The example below shows the use of the `__ramfunc` keyword.

```c
__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 */
        ...
        ...
    }
}
```
The critical parts of the linker file showing the memory placement and the various section initialization requirements are shown below.

```assembly
#define symbol FLASH_SIZE = 0x00001000; // 4 KB Erasable sector
#define symbol FLASH_BASE = 0x18000000; // 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 .intvec };  
do not initialize { section .mainstackarea };  
do not initialize { section .processstackarea };  

initialize by copy { rw };  

// IVT at start of flash
place at start of FLASH_region { ro section .intvec };  

// 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 };  
```

For the purposes of this example, we only wish the application code and data to be mapped to the first erasable 4 KBytes block in flash memory.

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 hand over to the application stored in flash memory.

Once the hardware initialization has completed, the routine must vector to the next application. For the purposes of this example, the next application has been mapped to SPI physical memory address 0x00001000, which correlates to the memory mapped address space of 0x18001000 and is the next non-occupied erasable sector of flash memory.

An example function responsible for the handover to the next application is shown below. This function takes a pointer to the applications vector table. The routine 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");
}

The application to be executed after the processor initialization sequence is a simple blink program. In order to bulk out the application, some data sections are included to make the application a little larger. This is simply intended to better highlight the advantages in the boot time by requiring more data to be copied from SPI 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. The blink routine is copied from the flash memory to SRAM space, where it is then executed. 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 sector.

// 2048k FLASH
#define symbol FLASH_SIZE  = 0x0001F000; // 2M byte QSPI Flash, first sector removed
#define symbol FLASH_BASE  = 0x18001000; // Internal stacked QSPI (SPI2) Flash

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 function __low_level_init() to their application that configures the hardware.

Firstly, 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, then 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 we also explicitly place them within a section using the location pragma as follows:

```c
#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 */
        ...
    }
}
```

The example implementation of the __low_level_init() function and the commands required in the linker file are highlighted below.

The function is preceded with two pragma commands. These allow for dedicated section operators to be used to determine the locations and sizes of the sections that we wish to copy and the sections that we wish to copy to.

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.

So in the previous example, if we were to initialize manually 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”.

The implementation below shows how we can manually copy all the data from flash memory to SRAM space in order to execute the code, all prior to the rest of the runtime initialization.

```c
#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;
}
```
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-CM403F 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 LED on the ADSP-CM403F 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.

<table>
<thead>
<tr>
<th>Hardware Initialization Method</th>
<th>Bring up time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize from main()</td>
<td>17.5ms</td>
</tr>
<tr>
<td>Multi Application</td>
<td>2.75ms</td>
</tr>
<tr>
<td>__low_level_init()</td>
<td>3.0ms</td>
</tr>
</tbody>
</table>

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 processors vector table from flash memory to SRAM space, prior to optimizing the system hardware. The system startup could be re-arranged to further optimize this process, however the benefits from such a task would be minimal. The Multi-Application implementation initially only copied over 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.

Application developers who are looking to implement In Application Programmers (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, thus having 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. The IAP can then use the techniques highlighted in the multi-application method to vector to the main application, if no firmware update is required.

Firmware update software is often required to be copied to SRAM internal memory before being executed, allowing for a firmware upgrade utility to update itself in flash memory. 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 on the main application bring-up time in the event a firmware update is not required.

References


Document History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
</table>