Separate Boot and Application Projects on C2000 Devices

Todd Mullanix

TI-RTOS Apps Manager

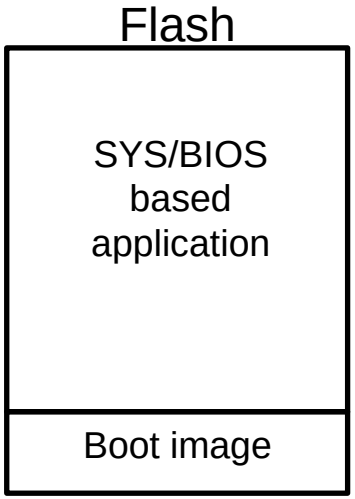
Oct. 9, 2017

Goal

This presentation will show how to have a small boot image and a SYS/BIOS based application both in flash for a C28 device. The boot image runs first and does a few things and then jumps to the SYS/BIOS based application.

Many customers like to have a separate boot image to manage different things (e.g. firmware updates, user interactions and diagnostics).

The basic idea is to have both the application and boot image in separate locations. When the device is turned on (or reset), the boot image runs first and then it jumps into the application (at `_c_int00`).



Examples

Along with this presentation are two projects that show this functionality. While for the F28377D Control Card, the concepts are applicable for other C28 devices.

- **typical**: Simple SYS/BIOS example that runs a task and outputs to the CCS console.
- **myC28Boot**: Simple project with one assembly file. It contains the reset vector that jumps to the typical application's `_c_int00` function.

Required Hardware to run examples

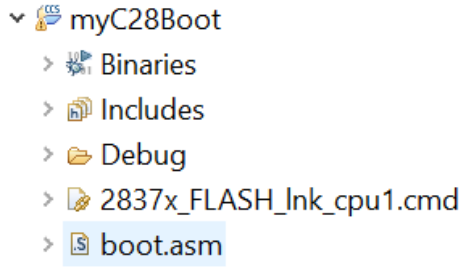
- F28377D Control Card.

Required Software to build examples

- SYS/BIOS: 6.50.01.12
- CCS: 7.x
- XDCtools: 3.50.3.33_core

Boot Project: Overview

Here is the boot project called myC28Boot. It has the linker file and a small boot.asm file.



This boot image does **not** use SYS/BIOS. It's purpose is to run first and do "stuff" and then jump into the application's `_c_int00` function to start the application.

The "stuff" is usually limited because you don't generally replace a boot image on a device. The "stuff" could be to manage which image to run based on some setting. Our boot simply turns an LED on and then starts the application. The LED is to prove that the boot image is running first!

Boot Project: boot.asm

The boot.asm is basically the same as F2837xD_CodeStartBranch.asm from ControlSuite.

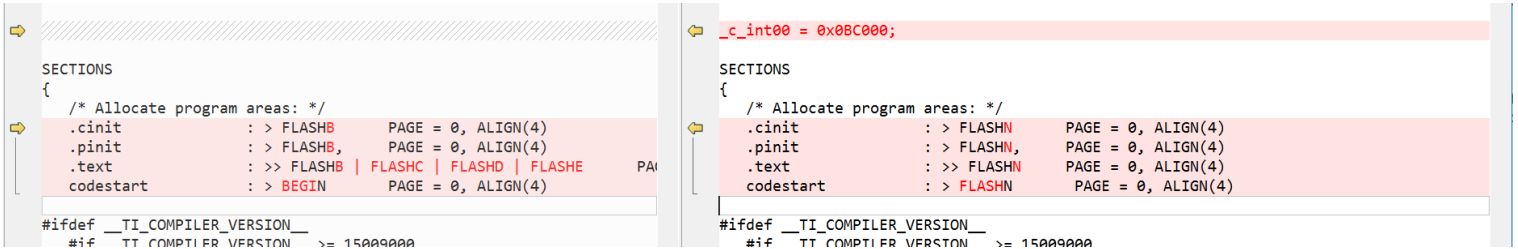
We'll make `code_start` be the entry point for the boot image.

LD2 is on by default on the ControlCard. Let's turn it off here to make sure this is working properly.

```
WD_DISABLE  .set  0    ;set to 1 to disable WD, else set to 0

    .ref _c_int00
    .global code_start

;-----------------------------------------------------------------------
* Function: codestart section
*
* Description: Branch to code starting point
;-----------------------------------------------------------------------

    .sect "codestart"

code_start:
    .if WD_DISABLE == 1
        LB wd_disable       ;Branch to watchdog disable code
    .else
        EALLOW
        MOVW DP, #0x01F0 ; Set datapage to GPIO Ctrl Reg base
        ZAPA
        MOV AH, #0x8000
        MOVL @0xA, ACC   ; Set GPIO31 to direction - output
        EDIS

        ; Turning off LED
        MOVW DP, #0x01FC ; Set datapage to GPIO Data Reg base
        ZAPA             ; Clear ACC
        MOV AH, #0x8000  ; Set bit 31 in ACC
        MOVL @0x2, ACC   ; Set bit 31 in GPASET (turns off LED)
        MOVW DP, #0x0    ; Restore datapage

        LB _c_int00         ;Branch to start of boot._asm in RTS library
    .endif
```

Boot Project: Linker File

This boot image does not do much. The standard linker file was changed to have all the code go into FLASHN. For the other application project, we'll not use FLASHN!



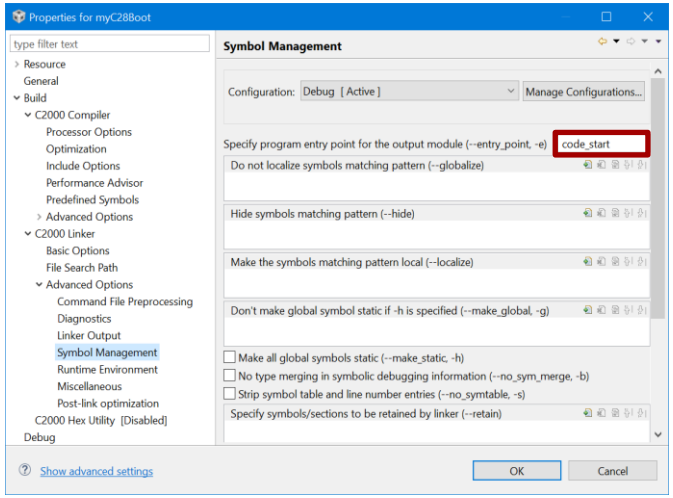
```
SECTIONS
{
   /* Allocate program areas: */
   .cinit              : > FLASHB      PAGE = 0, ALIGN(4)
   .pinit              : > FLASHB,     PAGE = 0, ALIGN(4)
   .text               : >> FLASHB | FLASHC | FLASHD | FLASHE      PA
   codestart           : > BEGIN       PAGE = 0, ALIGN(4)

#ifdef __TI_COMPILER_VERSION__
```

```
_c_int00 = 0x0BC000;

SECTIONS
{
   /* Allocate program areas: */
   .cinit              : > FLASHN      PAGE = 0, ALIGN(4)
   .pinit              : > FLASHN,     PAGE = 0, ALIGN(4)
   .text               : >> FLASHN     PAGE = 0, ALIGN(4)
   codestart           : > FLASHN       PAGE = 0, ALIGN(4)

#ifdef __TI_COMPILER_VERSION__
```

Additionally the linker file defines the `_c_int00` symbol that is used in the boot.asm file. We'll make sure the application project puts `_c_int00` at this address.

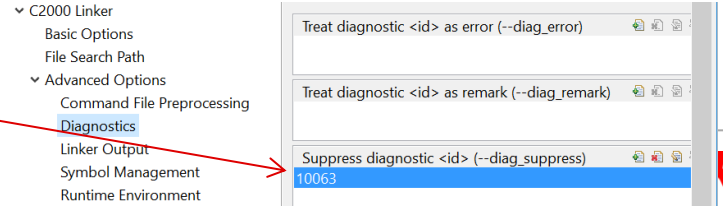
Boot Project: Project Settings

We need to make `code_start` be the entry point in the CCS project. This helps us when we load everything from CCS. In the boot project properties, add the new entry point.



If you build now, you'll get a build warning. You can get rid of this if desired by choosing to suppress it.

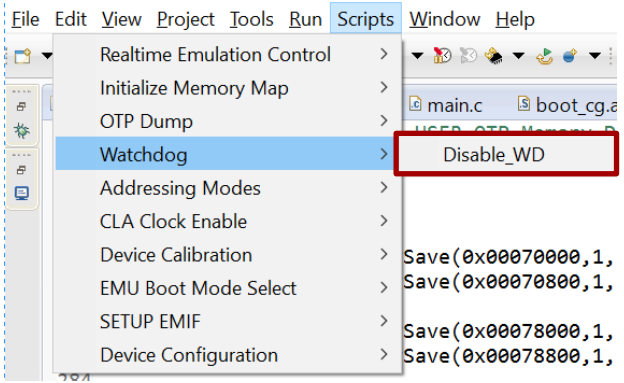
```
<Linking>
warning #10063-D: entry-point symbol other than "_c_int00" specified:  "code_start"
'Finished building target: myC28Boot.out'
```



Boot Project: Watchdog

Watchdog: If the `code_start` function was doing any real work, you might want to disable the watchdog or tickle it as needed.

When loading with **CCS**, the Watchdog might be disabled by CCS. Please refer to the gel file for your specific device. For the C28377, the f28377d_cpu1.gel file generates a hotmenu item called Disable_WD that can be used.



SYS/BIOS can enable/disable the watchdog also. For example for the F2837x devices, the following can be used to disable the watchdog.

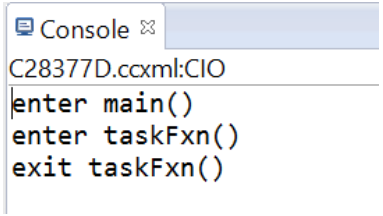
```
var Boot = xdc.useModule('ti.catalog.c2800.initF2837x.Boot');
Boot.disableWatchdog = true;
```

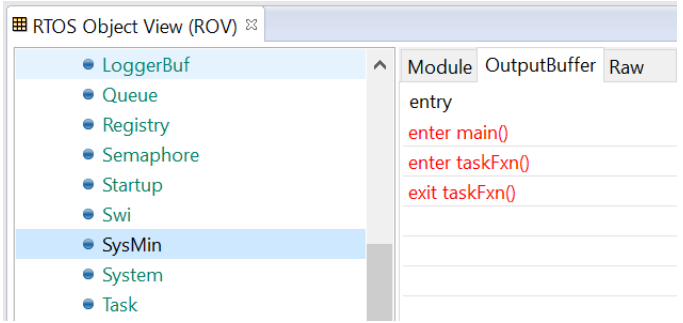
If enabled, it is up to the application to manage the tickling of the watchdog.

Application Project: Overview

The SYS/BIOS based application is one of the standard kernel examples. “Typical” has one task that calls:

- System_printf: Since SysMin is used in the .cfg file, the ASCII string is saved in an internal buffer*.
- Sleeps for 10 ticks: since we are using the default 1ms for a Clock tick, this means it sleeps for 10ms.
- System_printf: Again the ASCII string goes into an internal buffer.
- System_flush: Now the internal buffer is flushed to the CCS CIO buffer and shows up in the CCS Console
- After the System_flush, the app sleeps for a second and turns on the LED to denote the end of the app.





*The SysMin buffer can be viewed from Tools->ROV->SysMin. Shown here before calling `System_flush()` in `taskFxn`.

Application Project: Kernel Configuration Changes

The first section of code in the app.cfg file plugs in the new reset vector. "`&myBoot`" comes from the linker file (next slide). The next section of code places `_c_int00` at 0x0BC00 (FLASHN)*. Remember we used this address back in the boot image's linker file.

App.cfg

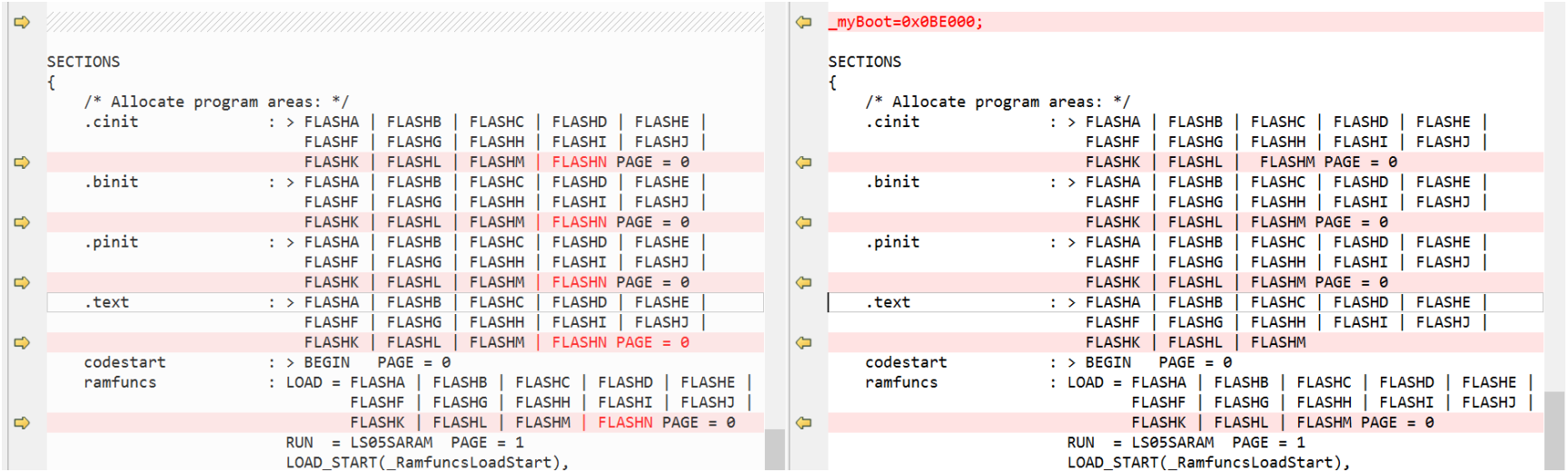
```
Hwi.nonDispatchedInterrupts[0] = new Hwi.NonDispatchedInterrupt();
Hwi.nonDispatchedInterrupts[0].fxn = "&myBoot";
Hwi.nonDispatchedInterrupts[0].enableInt = false;
Hwi.nonDispatchedInterrupts[0].intNum = 0;


Program.sectMap[".c_int00 { boot.a28FP<boot_cg.o28FP> (.text) }"] = new Program.SectionSpec();
Program.sectMap[".c_int00 { boot.a28FP<boot_cg.o28FP> (.text) }"].loadAddress = 0x0BC000;
```

*You can do this in the linker file instead if you prefer.

Application Project: Linker File Changes

Here's the before and after picture of the TMS320F28377D.cmd linker file in the application project.

- Remove `FLASHN` from all placements (not all removals are shown in the pic). We need to make sure that the application does not use it!
- Added symbol "_myBoot" symbol and set the value to the beginning of `FLASHN` (remember the boot image placed `code_start` here).



```
SECTIONS
{
    /* Allocate program areas: */
    .cinit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM | FLASHN PAGE = 0
    .binit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM | FLASHN PAGE = 0
    .pinit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM | FLASHN PAGE = 0
    .text               : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM | FLASHN PAGE = 0
    codestart           : > BEGIN   PAGE = 0
    ramfuncs            : LOAD = FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                                 FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                                 FLASHK | FLASHL | FLASHM | FLASHN PAGE = 0
                          RUN  = LS05SARAM  PAGE = 1
                          LOAD_START(_RamfuncsLoadStart),
```

```
_myBoot=0x0BE000;

SECTIONS
{
    /* Allocate program areas: */
    .cinit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL |  FLASHM PAGE = 0
    .binit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM PAGE = 0
    .pinit              : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM PAGE = 0
    .text               : > FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                            FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                            FLASHK | FLASHL | FLASHM
    codestart           : > BEGIN   PAGE = 0
    ramfuncs            : LOAD = FLASHA | FLASHB | FLASHC | FLASHD | FLASHE |
                                 FLASHF | FLASHG | FLASHH | FLASHI | FLASHJ |
                                 FLASHK | FLASHL | FLASHM PAGE = 0
                          RUN  = LS05SARAM  PAGE = 1
                          LOAD_START(_RamfuncsLoadStart),
```

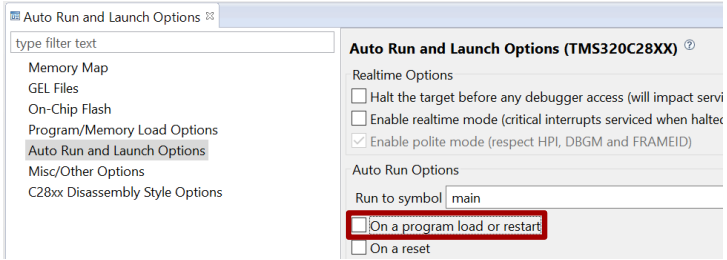
Application Project: Project Settings

<This page internationally left blank since there are no changes for this☺>

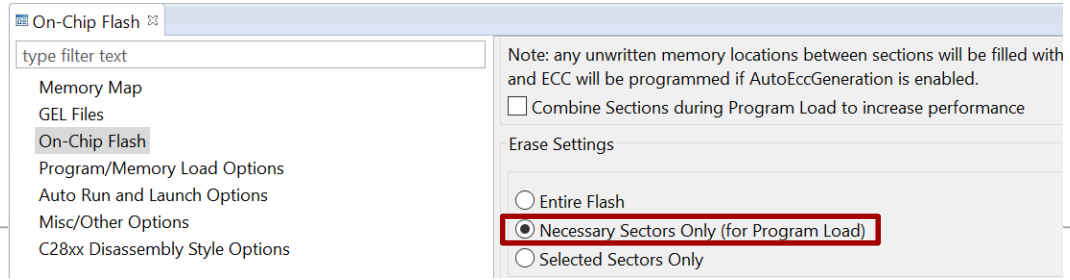
CCS: Loading

When using CCS you need to make a couple changes before you load the images.
Open **Tools->Debugger Options->Auto run and Launch Options**.

1. **Uncheck run to main**: This just makes it easier to see that everything is working. It also saves a hardware breakpoint☺

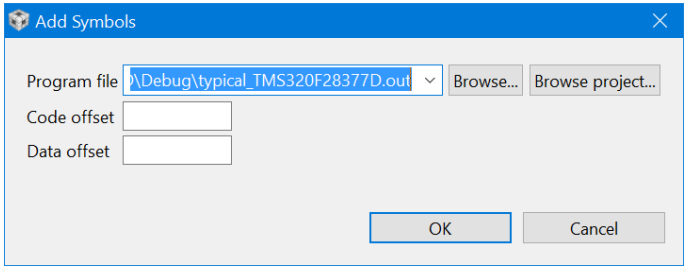


2. **Erase only necessary sectors**: When you load an image, only erase the sectors that are needed by the image. Otherwise the second load will wipe out the first one☹



CCS: Loading [cont.]

3. **Load the typical application image**: CCS knows the entry point is `_c_int00` for the application. You can run/debug the application as desired. The output should go to the CCS console (or you can look at it in ROV).
4. **Load the boot image**: CCS now thinks the entry point is `code_start`. If you start single stepping, you jump into `_c_int00`. It's hard to tell that though, so go to step 5.
5. **Add Symbols (optional)**: If you want to step into typical from the boot image and get source level debugging do "Run->Load->Add Symbols…" and select the typical image. Note: if you run, you'll get the CCS console output, but you cannot use ROV.



6. **Power Cycle/Reset**: The boot image will run (and turn the LED on) and then the typical application.

Application Project: Adding a Reset Function

Reset Functions in SYS/BIOS are called early in the booting of the application. Users can plug in their own function and perform actions. Note: the stack is not set up, so no local variables, calls to other functions, etc. The Typical example includes a reset function.

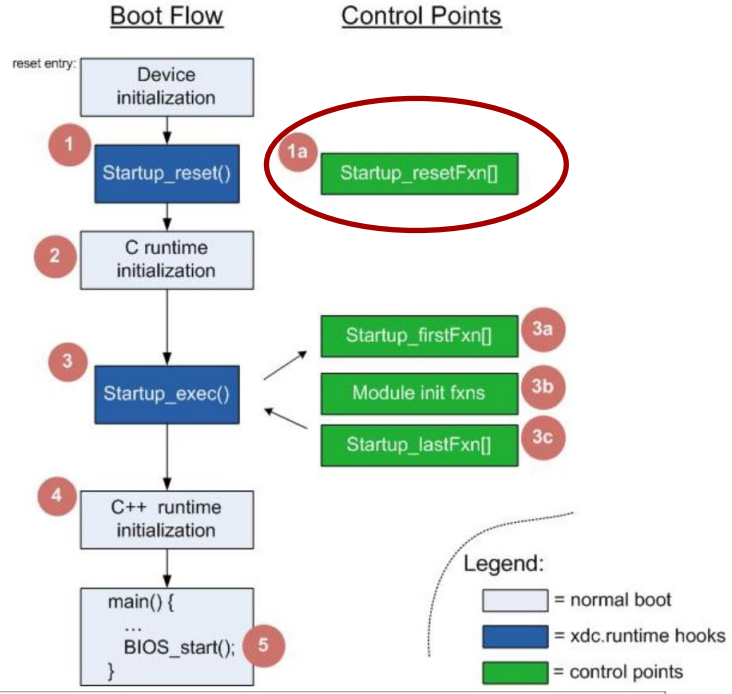
app.cfg

```
var Startup = xdc.useModule('xdc.runtime.Startup');
Startup.resetFxn = "&myResetFunction";
```

main.c

```
void myResetFunction()
{
}
```

The reset functions can be used instead of having a separate boot image in some cases. Diagram is from the first link in the “Additional Resources” Slide



Additional Resources

- http://processors.wiki.ti.com/index.php/SYS/BIOS_for_the_28x
- http://processors.wiki.ti.com/index.php/Interrupt_FAQ_for_C2000
- https://e2e.ti.com/

Appendix: Bugs/Enhancements

SYSBIOS-560: C28 Hwi module's plugMeta does not allow intNum 0

The Hwi.plugMeta function cannot be used in the .cfg since it is checking for zero. This would have simplified the .cfg code.

Jira tickets will be addressed in a future release.