Everything you always wanted to know about boot-loaders, and how to write one for an embedded application.

Embedded Apps Need Boot-loaders, Too

By Stanford Hudson

The term bootstrap, which is related to the old expression "pulling yourself up by your own bootstraps," means to accomplish something with minimum resources or advantages. A bootstrap loader, usually referred to as a boot-loader, is typically the first piece of software to execute in a system and is responsible for bringing the entire operating system to life. In an embedded environment, the first instructions are usually executed from slow, non-volatile memory. Therefore, one task of the boot-loader is to copy the main application from slow memory to a fast volatile memory before executing. Boot-loaders are also usually required to initialize data memory and system registers before jumping to the main application.

Boot-loaders come in many forms, ranging from the simple to the complex. The simplest form involves jumping to the main application code immediately from reset, without performing any system initialization or program loading. The main application is then responsible for setting itself up. More complex boot-loaders may perform system diagnostics, memory and system initialization, program verification, and loading before a final jump to the main application. Regardless of the complexity, nearly all boot-loaders, especially those in an embedded environment, can be a challenge to design and implement. Let's first take a look at a typical development environment and discuss some of the basic properties of boot-loaders.

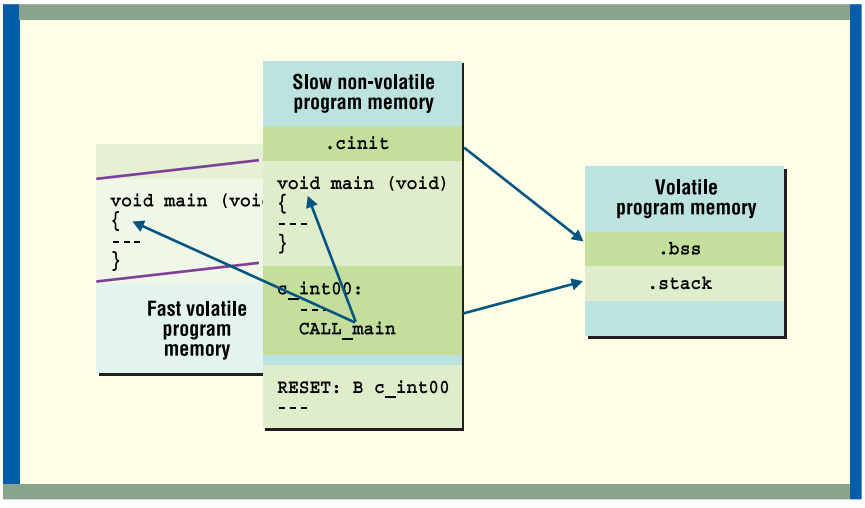

Figure 1. If the main program is required to run out of a faster memory device, c_int00 may optionally copy the program from slow memory to fast volatile program memory space before calling main().

Boot-Loader Basics

In a typical embedded development environment, target hardware is connected to a host via an emulator interface. Included in the environment is a debugger program, such as TI's Code Composer Studio, that enables the user to load and execute programs in target memory. The user can also set breakpoints, if required, and perform other tasks including real-time trace and profiling. This capability enables a developer to quickly realize and troubleshoot new applications.

If an application is written in C in

a non-embedded development environment, for instance, execution appears to begin at the main() function and the program initialization and loading phase is hidden from view. This is done to turn the attention of the user to the high-level main application code rather than the low-level tasks of initializing system registers, the stack, and data. It is simply the debugger's attempt to provide a user-friendly environment.

In an embedded development environment, however, execution usually starts at the C environment's program entry point. Sometimes labeled as c_int00, the entry point program is responsible for initializing the stack, system registers, and data memory before jumping to main(). It is assumed that the developer will be required to pay special attention to c_int00 since it will be invoked by a processor-specific reset entry point or vector.

Boot-loaders come in various forms. Figure 1 illustrates a typical boot-loader for an embedded C application. In this example, the RESET vector branches to the C environment's entry point function c_int00, where the stack and system registers are initialized. Additionally, C variables that require initialization before runtime are copied from the .cinit section and are written to their corresponding data addresses within the .bss section. Finally, the main() function is called. In this simple case, the boot-loader is comprised of the RESET vector along with c_int00, and it is assumed that the boot-loader and the main application both execute from the same non-volatile memory device. In a more complex system, the boot-loader may be faced with the task of copying the main application into fast volatile memory before calling main().

Boot-loaders may also perform other tasks such as system diagnostics, debug functions, and field upgrades. Diagnostics may include memory testing, peripheral testing, and program integrity verification. Simple debug functionality may also be included in a boot-loader, which is usually packaged into a monitor user interface. Monitor programs provide peek and poke, or "monitoring", capabilities as well as simple single-step debug functions, which may be useful when a board is under development. Boot-loaders may also handle field upgrades via in-circuit FLASH programming and program downloads from external memory sources.

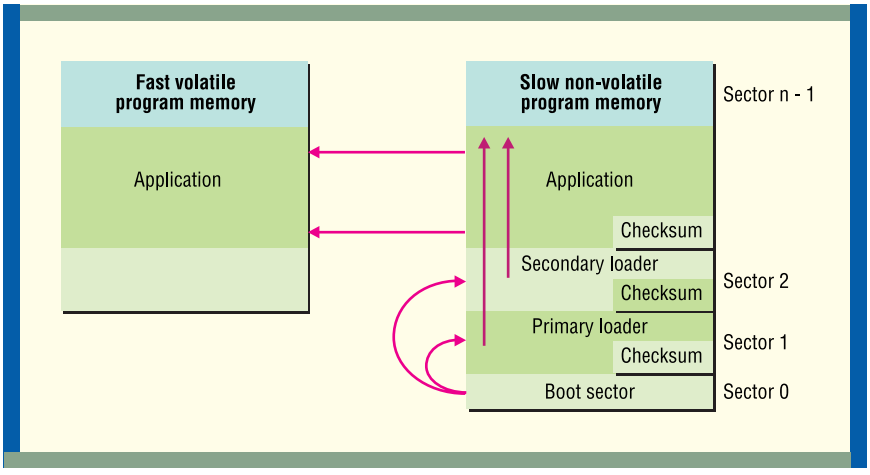

Figure 2. The boot sector invokes the secondary loader if the primary loader sector is damaged. The primary and boot sectors are locked permanently, while subsequent versions of the loader may be written into the secondary loader sector. In-circuit non-volatile memory programming may be interrupted without damaging the boot-loading system.

An Example

Let's look at an example boot-loader that incorporates all of the basic features. This example will also incorporate the copying of the main application program from slow non-volatile memory to fast volatile memory for execution.

The target processor is a TI TMS320VC5402 DSP, which utilizes separate program and data memory spaces. In this simple system, external memory is comprised of a slow FLASH device that contains the boot-loader program and a copy of the main application. We will use the C5402's overlay memory feature and place the DSP into microprocessor mode to enable us to use fast, internal DARAM for both programs and data.

The main application code is written in C. For this example, we will require the boot-loader to initialize the stack, set up a few of the C5402 system registers, initialize data from the .cinit section and then jump to main(). A modified version of the standard TI 5402 DSP library c_int00 function will serve as the boot-loader. The modified version will also copy the main application program external FLASH into DRAM before jumping to main().

The linker command file for this example is shown in Listing 1. We will section memory into four distinct spaces: VECT, EPROG, IPROG, and IDATA. Page 0 is used for program storage, while Page 1 holds data. VECT and EPROG both reside

in non-volatile memory, while IPROG and IDATA occupy fast internal DARAM. The VECT area, which starts at the C5402's default reset address, holds the interrupt vector table. EPROG contains both c_int00 and a copy of the C program. At runtime, the modified version of c_int00 copies the C program from EPROG to IPROG for execution. Finally, IDATA holds the stack and C variables in DARAM, which are initialized by c_int00. Note that in this example, we will set the OVLY bit to 1 in the PMST register on the C5402 to enable the DSP to share program and data space within the DARAM.

In the linker command file, an important point to note is the use of the LOAD and RUN keywords. When compiling any C or assembly source module, the resulting object file is always relocatable, therefore no absolute addresses are known until link time. An object file may be loaded in any area of memory temporarily, but must be copied to an absolute RUN address before execution. The LOAD keyword is used to indicate where a program should be stored, but not necessarily where it will be executed. The RUN keyword indicates the starting address of where to execute and is also used by the linker during address resolution. If the run and load addresses are different, this indicates that the program will be copied, at runtime, from the load address to the run address before execution. Normally, the run and load addresses are the same, but in this particular example, we must have separate addresses since the main application is stored in slow memory and must be copied to fast memory before execution. It is the responsibility of the boot-loader to perform this copy.

For the interrupt vector table, the reset vector simply branches to the c_int00 entry point function and no other interrupts are handled:

```
      .sect  ".vectors"
      .ref  _c_int00
RESET: B    _c_int00
      .end
```

The modified c_int00 entry point function must first initialize the

stack and then any required DSP registers. It also copies the main application program from FLASH to DARAM, initializes data from the .cinit table, and finally calls main() in DARAM. Note that we are not required to initialize the C5402's SWWR register since the application will execute out of fast, no wait-state internal DARAM. If the application was copied to an external non-volatile device, however, additional external program memory wait-states may be required.

The .text section mark, run, and length symbols in the linker command file are used to copy the main application from non-volatile program space to fast volatile program space. This is accomplished with just a few instructions:

```
.sect   ".text_mark"
.label  __text_load
.text
STM    #__text_run, AR1
RPT    #__text_length
MVPD   #__text_load, *AR1+
```

Special attention needs to be placed on where we store programs and data in the DARAM since they will be sharing the same memory in overlay mode. In general, programs may not be stored in overlay DARAM locations that are occupied by data and vice versa. As shown in the linker command file, the memory areas IPROG and IDATA, which occupy the C5402's DARAM, do not overlap.

Advanced Techniques

Nearly all TI DSP's contain internal ROM-based boot-loaders that may be used to transfer code from an external source to the DSP's program memory space. These boot-loaders are flexible and allow code to be downloaded through such peripherals as the McBSP, SPI, HPI, I/O, and the parallel memory interface. The mechanism by which these boot-loaders are invoked is device-dependent, but all usually involve the DSP detecting the digital state of an external processor pin or a data sequence on the external parallel data or I/O bus.

In most embedded environments, boot-loaders must be robust and, in some cases, completely fail-safe. One good design approach is to divide the boot-loader into independent sectors as illustrated in Figure 2. Boot-block FLASH devices are especially well suited for this type of arrangement since each sector can be locked individually.

The Boot sector, which contains the interrupt vector table as well as initial boot-up code, is programmed into the non-volatile memory space at the factory and never changes during. The loader is broken down into two sectors: Primary and Secondary. The Primary is shipped with the system and is typically the first revision of the loader.

The Secondary sector is left unpopulated. If a new release of the loader is required, it is written into the Secondary sector by the Primary loader. The Application sector contains the main application code, which is downloaded and written into non-volatile memory space by either the Primary or the Secondary loader during a field upgrade. The Primary loader, which is normally left untouched, can be re-programmed under strictly controlled conditions, as in a factory setting or during cold field upgrades. Upon system reset, the Boot sector performs rudimentary, hardware-specific system initialization, and, in general, is tightly coupled to the hardware. The Boot sector is also responsible for determining which loader to invoke based on an integrity check (CRC or checksum) that is stored for each loader sector. The integrity check involves computing the checksum and verifying that it matches the stored checksum. A checksum mismatch implies the sector has been damaged or is

Listing 1: Linker Command File

```
MEMORY
{
    PAGE 0: IPROG:  origin = 0x0800 length = 0x0800
    PAGE 0: EPROG:  origin = 0xC000 length = 0x0800
    PAGE 0: VECT:   origin = 0xFF80 length = 0x0080
    PAGE 1: IDATA:  origin = 0x0080 length = 0x0780
}
SECTIONS
{
    .cint00:  load = EPROG
    .bss:     run  = IDATA
    .stack:   load = IDATA
    .cinit:   load = EPROG
    .vectors: load = VECT
    .text :   load = EPROG, run = IPROG
    {
        __text_run = .;
        *(.text_mark)
        *(.text)
        __text_length = .- __text_run;
    }
}
```

unpopulated. Subsequently, the Loader sector is responsible for verifying the application sector checksum, loading the application into volatile memory, and jumping the main application entry point.

Some FLASH-based embedded targets may require some type of field upgrade capability. In-circuit programming involves writing to a non-volatile memory device while it still resides in the target system. One approach is to first download a device loader program to non-volatile target memory. The device loader is subsequently responsible for downloading application program data from an external source and programming that data into non-volatile memory on the target. The ROM-based boot-loaders found

in most TI DSP's enable the user to easily perform this task.

Another common problem for DSP-based boot-loaders is the handling of constants. Due to DSP memory architectures, constants that are stored in application code space must be copied to data space before they are accessed. This is accomplished by defining the load address of the .const section to reside in nonvolatile program memory with the run address being in volatile data memory. The mark address and length symbols are computed in the SECTIONS directive for the definition of the .const section:

```
.const: load = EPROG, run = IDATA
{
  __const_run = .;
  *(.const_mark)
  *(.const)
  __const_length = .- __const_run;
}
```

The .const mark and length symbols are then used by the boot-loader to copy constants into data memory in a similar manner as used previously for copying programs in the boot-loader example:

```
        .sect  ".const_mark"
       .label __const_load
       .text
const_copy:  LD   #__const_length, A
       BC   end_const, AEQ
       STM   #__const_run, AR2
       RPT   #__const_length-1
       MVPD  #__const_load, *AR2+
end_const:  RET
```

In some embedded systems, the amount of non-volatile memory may be considerably smaller than volatile memory. ◆

Stanford Hudson (shudson@tekgenix.com) is a member of technical staff at Tekgenix Corporation, a consulting and design firm located in Richardson, Texas that specializes in DSP-based hardware and software design, analysis, and rapid prototyping. He has 11 years experience in real-time embedded systems architecture. He previously worked in the digital cross-connect software group at Alcatel USA.