Overview of the Linux Runtime Sharc Loader on the ADSP-SC58x Processors

Contributed by Yi, Gabby

Rev 112 – May 25, 2017

Introduction

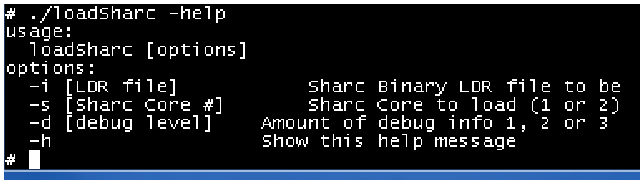
The ADSP-SCxx processors are comprised of an ARM Cortex A5 core alongside dual DSP SHARC+® cores. Analog Devices, Inc. provides a Linux Add-In BSP that allows a user to run Linux on the ARM core. When a user wants to run an application on one of the SHARC+® cores, a debugger, such as the ICE-2000 can be used to load the SHARC+® cores without disturbing the ARM core running Linux.

When a SHARC+® application is ready for deployment, the user must create a single boot stream containing u-boot and the SHARC+® application for each SHARC+® core.

This EE-Note describes a method to load different boot stream loader (LDR) files from the Linux filesystem to run on the SHARC+® cores during runtime.

Runtime SHARC+® Loader

The Runtime SHARC+® Loader (RSL) is a user Linux application that reads binary boot stream LDR file and boots it onto one of the SHARC+® cores. Below in Figure 1, the help message is displayed for the RSL. By default, if no SHARC+® core is specified, core #1 will be the target.



```
# ./loadSharc -help
usage:
  loadSharc [options]
options:
  -i [LDR file]          Sharc Binary LDR file to be
  -s [Sharc Core #]      Sharc Core to load (1 or 2)
  -d [debug level]     Amount of debug info 1, 2 or 3
  -h                   Show this help message
# 
```

Figure 1- Help message from Runtime SHARC+® Loader

There is another option "-d" to allow the user to specify the debug level for the amount of information displayed during the execution.

Architecture

The RSL is comprised of a front-end and a back-end. The front-end is the Linux application that reads in the binary boot stream LDR file. The back-end is a Loader Stub (LS) which runs on the SHARC+® core and communicates with the RSL front-end. The RSL is also responsible for loading the LS onto the SHARC+® for execution.

The LS employs the use of the boot kernel stored in boot rom memory to do the actual booting of the boot stream. It supplies a "Load" function driver to be registered with the boot kernel to be used during the booting. As the booting occurs, the supplied "Load" function is called to fetch more boot stream data. Since the LDR file sits on the file system on the ARM core, the "Load" function communicates with the RSL front-end how much data to fetch. The RSL front-end will act as a server and wait for this request. Once it gets a request, it'll `fread()` the requested amount from the LDR file and the store it in the shared buffer with the SHARC+® core.

There are two main reasons why the RSL was architected this way.

1. It's easier for the boot kernel to load and boot a boot stream then for a Linux application to parse and load a DXE image file.
2. The boot kernel API is not Linux compatible so a Linux application cannot directly employ the boot kernel API to boot a LDR file.

Operational Flow

In Figure 2, the diagram pictorializes the flow of the SHARC+® application loading. The following is the sequence of steps which occurs.

1. Set `RCU_SVECTn` register to location in boot rom to loop on idle instruction.
2. Reset the SHARC+® core.
3. Load the Loader Stub onto the SHARC+®'s L1 memory.
4. Set the `RCU_SVECTn` register to the beginning of the Loader Stub application in SHARC+® L1 memory.
5. Reset the SHARC+® core.
6. The RSL front-end waits for signal from LS that it's running.
7. The LS starts executing and sets up the boot structure of type `ADI_ROM_BOOT_CONFIG` and calls `adi_rom_BootKernel()` to start the boot procedure.
8. The RSL waits for a request from LS for boot stream data
    a. Once request is obtained, the RSL reads data in from LDR file and stores it in shared memory.
    b. The RSL signals to LS that boot stream data is ready.
9. This loop continues until the booting finishes and the LS signals to the RSL that no more data is needed.
10. RSL exits and LS calls the booted application as a function call to start execution.

Overview of the Linux Runtime Sharc Loader on the ADSP-SC58x Processors

Contributed by Yi, Gabby

Rev 112 – May 25, 2017

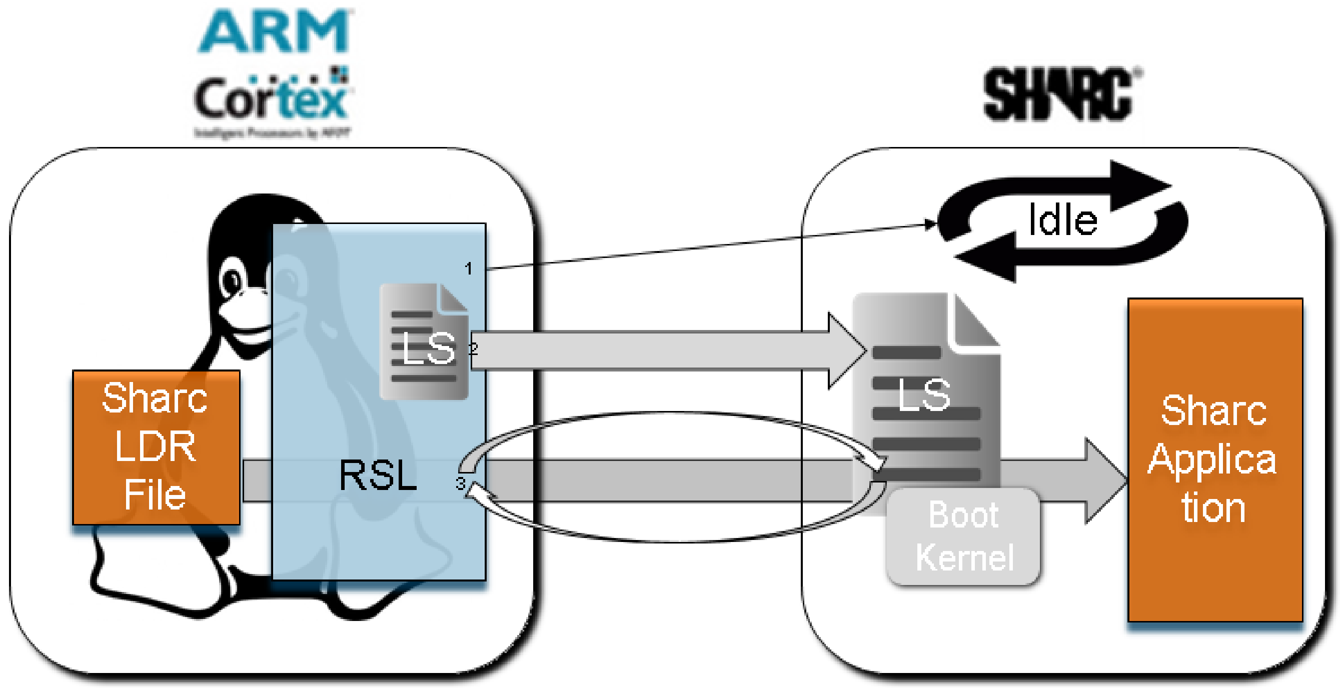


Figure 2-Linux Runtime SHARC+® Loader Block Diagram

Loader Stub

The Loader Stub is a SHARC+® application. It does two things. First, it sets up the boot configuration structure that that the boot kernel and then using the boot API, calls the boot kernel. Second, it provides "Load" function to be registered with the boot kernel. The LS is essentially doing a memory boot. The boot API allows users to define their own drivers to support custom boot modes. Since memory boot is not one of the supplied boot modes, a user would need to supply a custom driver. A driver is a set of functions which include, "Init", "Config", "Load", and "Cleanup". For a running system, memory is already configured and initialized, so there would not be a need for any of the other functions other than the "Load" function.

More information about the boot rom and boot kernel can be found in [1] and [2].

Combining Loader Stub with RSL Linux Application

As described above, the RSL loads the Loader Stub into SHARC+® memory for execution. Before this happens, the LS is compiled into RSL. For this to happen, using a script and `ELFDUMP.EXE`, which is a utility that comes with CrossCore® Embedded Studio (CCES), the contents of the Loader Stub DXE's are dumped and parsed and formatted into C source code data buffers.

From the provided source code, `SharcBooter_Core1` is the CCES project for the Loader Stub for Core 1 and `SharcBooter_Core2` is the CCES project for the Loader Stub for Core 2.

A provided script `makeSectData.sh` outputs `secdat_sh1.c` and `secdat_sh2.c` which can be compiled into the RSL.

Loader Stub Memory Placement

Since the Loader Stub is a SHARC+® application running from SHARC+® memory and it will be calling the boot kernel to boot another SHARC+® application that will also reside in SHARC+® memory, care must be taken not to overwrite the Loader Stub during the booting. So for this reason, the Loader Stub is placed in the upper portion of L1 Bank 3 memory at **0x0039A000**.

Since the LS resides in the upper portion of L1 Bank 3 memory, the SHARC+® applications to be loaded by the RSL should **not** use memory at **0x0039A000** and above.

Loader Stub Size

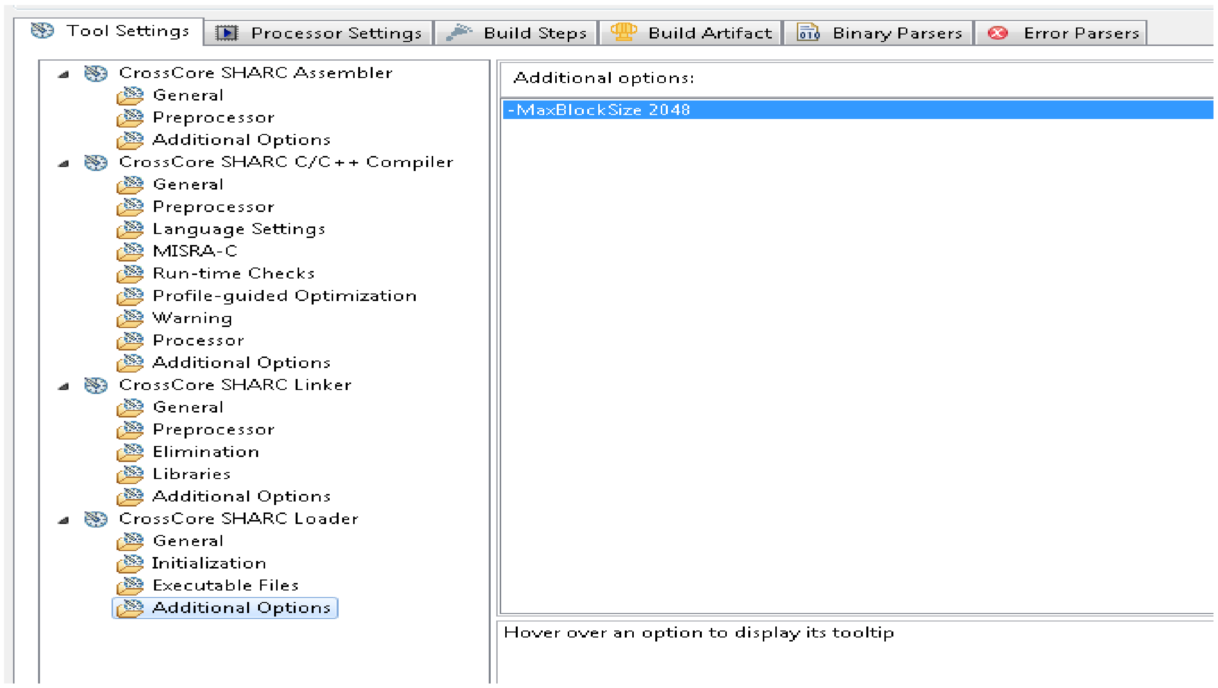
The size of the Loader Stub is primarily dominated by the shared memory buffer that the RSL front-end places data into from reading the LDR file and where the LS reads from when the boot kernel requires more boot stream data.

For this implementation, the buffer size is defined to be 2KB. Due to this, the user must restrict the block size of the boot stream of the SHARC+® application to be loaded to 2KB.

The maximum block size for a block in the LDR file of the SHARC+® application must be equal or less than 2KB.

To do this, the user can provide an extra option for the CrossCore® SHARC+® Loader utility. In the tools settings for the SHARC+® CCES project, there is a sub-section for `Additional Options` under CrossCore® `SHARC+® Loader` section. In this, add the switch `–MaxBlockSize 2048`.

A snapshot of this is shown in Figure 3 below to show this procedure.



Additional options:

```
-MaxBlockSize 2048
```

Figure 3 - Specifying Maximum Block Size in LDR File

Loading the Loader Stub

Once compiled into the RSL, the loader stub is simply copied into the correct SHARC+® memory locations. But, before this can be done, it must be ensured that the SHARC+® core isn't trying to access the same memory locations at the same time. The RSL doesn't know what state the SHARC+® core is in. It might already be executing an application which share the same memory locations as the Loader Stub. Therefore, the RSL sets the `RCU_SVECTn` register for either SHARC+® core 1 or core 2 to a location in read-only boot rom which hold instructions for looping on an "`idle`" instruction. Then the Reset Control Unit (RCU) is used again to trigger a core reset which will make the SHARC+® core jump to the location specified in the `RCU_SVECTn` register. Now, the RSL is guaranteed that the SHARC+® core is in a safe state for which the RSL can load the Loader Stub.

Communication and Handshaking

Loading SHARC+® with Loader Stub

On the ADSP-SC58x, the SHARC+®'s internal memory is accessible by other system masters which include other cores via the multi-processor address space.

When the Loader Stub contents are dumped and parsed, `secdat_sh1.c` and `secdat_sh2.c` are created, the memory addresses refer to the SHARC+®'s internal memory space. For the RSL to loads content into SHARC+® memory, these memory addresses are translated to multi-processor space addresses.

Boot Stream Data Buffer

As described earlier, the LS declares a buffer which is used by both the LS and the RSL. When `secdat_sh1.c` and `secdat_sh2.c` are created, the script also obtains the SHARC+® address for this data buffer to load into a pointer variable declaration for the RSL. This way, the RSL knows where to place the boot stream data.

Semaphores

Using the same method, a pair of volatile variables are declared in the LS. The addresses are then parsed and then loaded into variable declarations in the RSL. One of the variables in the pair is used by the SHARC+® to write to and signal to the ARM core. This *shared* variable is then used only as a *read-only* variable by the ARM core. This is done so there is no possible contention between both cores trying to write to the same memory location (variable). The other variable in the pair is used in the same fashion but the other way. The ARM uses it to write to and communicate to the SHARC+® core and the SHARC+® core uses it only as a read-only variable.

Accessing Physical Memory

Thus far, the mechanism by which the RSL front-end (ARM core) and the SHARC+® communicate and share data with each other. This is primarily by accessing the SHARC+® memory by translating it to the multi-processor address space. Basically, this is done by taking the SHARC+® byte address and then pre-pending 0x280 to the most significant ten bits of the SHARC+® L1 address if using slave port 1 or preprend 0x281 if using slave port 2. For this implementation, slave port 1 is used. For more information on multi-processor space addressing, refer to [3].

Still, Linux cannot just access these memory addresses directly. Linux runs in a Virtual Memory space, and any of the address provided by the parsed output of the LS binary image are physical addresses. In order to access this memory, the a physical address region is memory mapped using `mmap()`. The result is a virtual address which corresponds to the start of the physical address region. From this virtual start address, another address translation is calculated.

Details of this process is outside the scope of this EE-Note and will be provided in a separate upcoming EE-Note.

SHARC+® Application Boot Stream Constraints and Further Investigations

Memory Placement

As noted previously, the LS resides in a section of L1 memory and calls the boot kernel API to boot the SHARC+® application and then performs a direct call to the application. Therefore, the SHARC+® application cannot use the same memory region as the LS otherwise the LS would be overwritten before booting completes.

Moving the LS to L2 memory maybe a possible solution which will be investigated.

Cache

Currently, cache is also non-functional. A SHARC+® application which uses cache (instruction or data) will boot but then stall during cache initialization and set up.

System Interaction

There are system interactions that need to be considered as well. The RSL only resets a SHARC+® core and boots a SHARC+® application during a running system. If a previous SHARC+® application configured peripheral to run and use DMA, there might still be SHARC+® memory accesses ongoing while the RSL is trying to load the LS into SHARC+® memory. As such, there are certain situations where the RSL is prevented from loading subsequent boot stream LDR files.

Example Code and Projects

Accompanied with this EE-Note is a .zip file that contains three other .zip files. The first is RSL.zip. This is the source code for the user Linux application, otherwise referred to as the RSL front-end. The Loader Stub source data is already parsed and ready to compile into the RSL. A makefile is also provided to use the GCC toolchain provided in the Linux Add-in to help compile the RSL.

Finally, the two other .zip files are windows CCES projects, `SharcBooter_Core1` and `SharcBooter_Core2`. These are the projects for the Loader Stubs. Both projects use a script provided in their project directories called `makeSectData.sh`. This is a bash script that was used and run under Cygwin to run elfdump.exe from CCES along with SED and other utilities to parse the output DXE to create `secdat_sh1.c` and `secdat_sh2.c`. The script run by providing the path to the DXE image and which core it will run on. Future work would be to port this to Python and rid the requirement of Cygwin

The RSL can be executed as such:

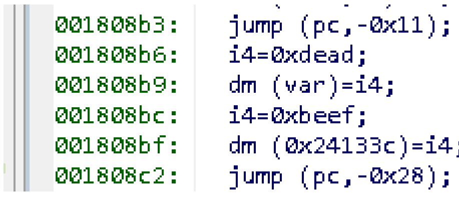
```
./loadSharc -i blinky_Core1.589.ldr -s
1 -d 3
```

Listing 1- Running the RSL

Above in Listing 1 shows how to run the RSL. The *-i* switch provides the input binary LDR file to be booted. The *-s* switch indicates which SHARC+® core to boot and the *-d* indicates the verbosity level of information printed out to console.

There are three sample LDR files provided in sharcLoader.zip; two LED blink programs, one for each core, and one talkthru program that runs on SHARC+® core 1.

The LED blink program contains code that loads two locations of a global buffer with 0xDEAD and 0xBEEF. Besides verifying that the LEDs blink on the EZ-Board, a user can load a "symbols only" session on CCES and see the following in the disassembly window.



```
001808b3:    jump (pc,-0x11);
001808b6:    i4=0xdead;
001808b9:    dm (var)=i4;
001808bc:    i4=0xbeef;
001808bf:    dm (0x24133c)=i4;
001808c2:    jump (pc,-0x28);
```

Figure 4 - Disassembly Window of Instructions in LED Blink Program

The accompanied code has been developed on an ADSP-SC589 EZ-BOARD. The described framework should work across the ADSP-SC58x family of processors.

Conclusion

This EE-Note demonstrated a method to boot different SHARC+® boot streams from Linux running on the ARM core of the ADSP-SC58x during runtime.

The method employed numerous techniques to solve issues such as:

1. Accessing physical memory from Linux
2. Resetting the SHARC+® core
3. Assessing SHARC+® memory
4. Putting the SHARC+® Core in a safe state while loading SHARC+® memory
5. Using the boot kernel stored in the boot rom

References

[1] *ADSP-SC58x/ADSP-2158x SHARC+®+ Processor Hardware Reference (http://www.analog.com/media/en/dsp-documentation/processor-manuals/SC58x-2158x-hrm.pdf) . Rev 0.4, Feb 2017. Analog Devices, Inc.*

[2] *Engineer-to-Engineer Note 384 : Tips and Tricks Using the ADSP-SC58x/ADSP-2158x Processor Boot ROM (http://www.analog.com/media/en/technical-documentation/application-notes/EE384v01.pdf). Rev 1, Sept 30, 2015. Analog Devices, Inc.*

[3] *Datasheet: SHARC+®+ Dual Core DSP with ARM Cortex-A5 (http://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-SC582_583_584_587_589_ADSP-21583_584_587.pdf). Rev 0, Oct 2016*

Document History

Revision	**Description**
Rev 0.0 –May 9th, 2017 byG.Yi	Initial Draft Release
Rev 0.1 – May 18th, 2017 by G. Yi	Removed parity checking restriction. It works now.
Rev 0.2 – May 24th, 2017 by G. Yi	Updated title