Serial Boot
An alternative and effective way of booting SoC externally.

Ankur Sharma
Design Engineer-MSG
Freescale Semiconductor

1 Introduction:
SoC booting involves initial set of operations that needs to be performed once the SoC comes out of reset. It involves the initializing of SoC resources like memory configuration, clock settings etc which prepares the SoC for functioning and then passes control to the application code. Once out of reset SoC looks for boot code at defined locations depending on SoC architecture.

Common boot options available in SoCs are internal, serial, and external boot mode. Depending on the boot mode, the appropriate minimum MCU resources are initialized to start user application code, this is achieved by a hard coded logic or a ROM code within the SoC. Internal booting provides the boot code fused in the internal memory. External booting involves boot code provided in external memory accessed through the SoC external bus interface. Of all the external booting options, Serial Booting is very commonly used.

Serial booting uses the serial protocols available within the SoC like LIN, UART, CAN, SPI etc to get the bootable code from another device and then execute the code. The code can be used to modify the original boot code being fused in the internal flash also.

This article deals with the Serial boot protocol and preparing a boot code for the serial booting. It tells about the things to take care while generating the boot code and the process of booting up the device using serial protocol. LIN and CAN are the common communication protocols being used in the automotive MCUs for serial booting.

2 Concept of Serial Booting:
Entering into serial boot mode can depend on SoC architecture. The SoC can have special boot configuration pins which decide the boot option on power up or it can have an algorithm implemented to jump to serial boot mode if other boot mode fails.

If a SoC provides various serial buses for booting then which bus has to be used for booting can be selected by either of the two ways:

1) Provide separate boot config pins options for each serial bus
2) SoC snoops receive pins of each serial bus and the first one to receive valid data externally is selected for booting. In this case all bus pins have to be configured and once data is received on a bus, the other bus pins are sent to their default reset values.

In either case when serial boot mode is selected the SoC has to configure its resources to receive the boot code from the other device. This includes setting up the serial bus pins for receive mode and configuring the protocol registers so that data can be received properly. There is a piece of hardcoded logic embedded in the SoC which takes care of serial booting. The major operations performed by this code are pin configurations, protocol register setting, copying the data to internal memory and after checking the validity and security options give the control to the downloaded code. With increasing speed of less complicated serial protocols, it is possible to execute the code from the serial interfaces directly without having to copy them in the internal RAMs.

3 Serial Booting Protocol:

Once the serial connection is established between the SoC and HOST device, the following protocol steps have to be followed

1. Security and password clearance
2. Send the code information like size, format and address where the code has to be stored
3. Download the application code
4. Switch to the downloaded application code if valid code has been received

**STEP1:** The SoC must provide some security features for the serial booting so that any hacker can not simply put a patch over the original fused boot code and change the functioning of device. The common security feature is to provide a password which has to be provided before serial booting. This password is matched with internally stored password and only if it matches the SoC receives the external boot code. More advanced security features like crypto engines, security algorithms provide sophisticated mechanism to guard against unauthorized accesses and temper protections. SoC’s can have limited access permissions for example it will allow the external code to only access the internal SRAM and not the fused code in internal flash. In this way the original fuse code remains intact.

**STEP2:** The SoC has to be provided with the information like the size of the external boot code and the START ADDRESS of internal memory where the code has to be dumped. The SoC keeps a track of the data being copied using an internal counter and when the specified size of the code has been copied, the SoC jumps back to the START ADDRESS

**STEP3:** Each byte of data received is stored in the MCU memory, starting at the START_ADDRESS specified in the previous step. The data increments through memory until the number of bytes stored matches CODE_LENGTH specified in the previous step.

**STEP4:** Once the code has been downloaded, the serial bus pins are reverted back to their reset values. SoC passes control to the loaded code at START_ADDRESS, which was received in step 2 of the protocol.
**Additional features:**
SoC may also provide additional features

1) **Watchdog:** The internal watchdog starts when serial booting protocol is selected. It is refreshed when the external code is written into the internal memory. The watchdog will give a timeout reset if the code bytes are not received within the specified time. It protects the SoC from being stuck in the serial boot mode.

2) **ECHO Operation:** The SoC can also have the feature of echoing back the data which has been received so that the external device can validate if the data being sent and received by the SoC are correct.

## 4 Serial Protocol Settings:

SoC configures the serial bus resources for serial booting. Apart from pin configuration, protocol settings have to be done for proper reception. One of the things to decide is the data transfer rate for code transfer.

For example for CAN and LIN serial mode, the receive pin configuration and baud rate settings are required before the data transfers begin. There are two options for the baud rate decision:

1) **Fixed Baud:** Always transfer the code on a fixed baud rate depending on the SoC clock

2) **Autobaud:** Before sending the password information, a dummy packet is send. SoC samples this packet and configures the protocol according to the baud rate of the received packet. Now the next incoming packets will be sampled properly.

**NOTE:** Since MCU will take some time to do calculations and set the CAN and LIN parameters some delay might have to be induced between zero byte and password byte

The communication is done in half-duplex manner; any transmission from host is followed by the SoC transmission. The host computer should not send data until it receives echo from the SoC.

Additional constraints can be put on protocols for more secure serial boot transfer. For example if CAN is used for serial download, the data is packed into standard CAN messages. For each frame like password, code info, application code etc, the CAN frame IDs can be fixed which will mean that serial frames having the specified data will be copied and rest will be ignored.

## 5 Application Code Generation:

### 5.1 Write the Boot code:

The application code to be uploaded in the SoC should be comprehensive enough to do the SoC settings required for the application. The whole boot code which is generally written when booting from flash should be part of serial boot code also. Since this will be loaded in
the internal memory of the SoC, the serial boot code is limited by the size of the internal memory.

**Note:** Since SoC starts loading the code at the start location specified and jumps to the start location after code is downloaded, there should be a valid code written in that location. So, using the linker script makes sure that boot code starts at the starting location itself.

### 5.2 Fill up the Memory Holes:

The SoC will receive the data and place it serially in the internal memory. The code may have different code portions like stack configured at different memory location and interrupt vectors at different memory locations apart from the normal text code. These different locations are specified through the linker file of the code. SoC is unaware of code partitioning and it will keep placing the data at next location serially starting from the START location mentioned as the first step.

For example if the data is first written in 0x40000000 till 0x400000FF and the next data is written in 0x40000B00, SoC will still write the data at location next to 0x400000FF. So it is important to fill the holes so that the code gets placed at desired location.

All the compiler tools give options for filling the memory holes. Below is a snapshot of compiler tool.

![Argument for filling memory holes](image)

### 5.3 Convert Binary data to Hex data:

The binary file generated after memory holes needs to be converted to Hex data so that it can be easily transferred within serial protocol transfer formats for example LIN or CAN frames, for SPI data packets.

Any commonly available utility can be used to convert the binary file into the Hex data file.
5.4 Creation of C-array:
The hex data generated in previous steps is included in the code of HOST which drives the serial bus of the SoC in the form of a C-array for example

Unsigned char eprom[] =
{
   0x71, 0x9F, 0xE7, 0xF3, 0x71, 0x90, 0xC0, 0x00, 0x51, 0x6C, 0x00, 0x00, 0x75,
   0x6B, 0x00, 0x3D, 0x55, 0x6C, 0x00, 0x00, 0x70, 0xE0, 0x00, 0x00, 0x70, 0xE0,
   0xC0, 0x00, 0x7C, 0x53, 0xA6, 0x7C, 0xC0, 0x00, 0xA6, 0x70, 0xC0, 0xD2,
   0x00, 0x24, 0x00, 0x02, 0x01, 0x15,
}-----------------------------------------------------------------------------------------------

5.5 Transfer of C array using LIN or CAN:
The C array generated in the above steps contains the code to be loaded in the DUT memory at required addresses.

The HOST code can now read the data from the C-array and write it in the LIN or CAN frames. In this way the SoC boot code in form of C array gets transferred in LIN or CAN frames.

6 Validating the Serial Booting:
Oscilloscope can be used to validate the data being transmitted by the host device and the data echoed back by the SoC. The following snapshots were taken while serially booting the 32 bit microcontroller providing the echo and Autobaud features in serial booting protocol using LIN or CAN options.

Snapshot below shows the Tx and Rx data lines of the LIN of SoC. LIN has been configured in the UART mode for easy of use. As the valid data is received on the Rx line, the data is echoed back on the Rx line after some delay.
Application code generation for Serial Booting

Note: In case of autobaud rate serial booting, zero byte frame is send before the password frame and once the SoC configures protocol the password frame is echoed back on Tx Line. Figure shows the autobaud feature in CAN serial booting.
7 Test Hardware Setup:

- Example Hardware setup with devices having CAN and LIN serial buses available for serial boot can consist of two boards, one functioning as HOST and other as SoC in serial boot mode. Both the boards can be of same devices or HOST device can be a different SoC having LIN and CAN
- SoC is kept in serial boot mode using the bootcfg pins
- In the HOST device the test code is loaded into SRAM and using the debug software the application code is loaded into the DUT by running the code on HOST device.
- The CAN and LIN pins can be connected through a transceiver or the LIN pins can be even directly connected without transceiver.

8 Conclusion:

Serial booting is limited by the slow speed of serial protocol and downloaded code size is limited by the size of internal memory. But it provides a big advantage of booting up the SoC from other device when no external memory is available, with the use of simple serial protocols like LIN, CAN, SPI etc. The hardware configuration to be done in the SoC for implementation of serial booting is very minimal and can be easily provided. If some issues are seen in the boot code programmed in internal memory then the serial booting will allow the manufacturer and authenticated user to debug the issues, modify the code and put a patch for the correct functioning.