Application Report
SPNA196–September 2013

# SPI Bootloader for Hercules RM48x MCU

*Quingjun Wang*

**ABSTRACT**

This application report describes how to communicate with the Hercules™ serial peripheral interface (SPI) bootloader. The SPI bootloader is a small piece of code that can be programmed at the beginning of Flash to act as an application loader as well as an update mechanism for applications running on a Hercules Cortex™-R4 based RM48x microcontroller.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spna196.

**Contents**



**List of Figures**



**List of Tables**



## 1 Introduction

This application report describes how to work with and customize the Hercules sample bootloader application. The bootloader is a small piece of code that can be programmed at the beginning of Flash to act as an application loader as well as an update mechanism for an application running on a Hercules microcontroller. The default build of the bootloader is compiled to use the SPI2 interface for updating either the application or the bootloader. The bootloader is provided as source code, which allows any part of the bootloader to be completely customized. The bootloader was built and validated using Code Composer Studio™ v5 and the RM48x Hercules Development HDK.

A visual C++ project for the PC is provided to download an application via the bootloader.

Table 1 shows an overview of the organization of the source code provided with the bootloader.

**Table 1. List of Source Code Files Used in SPI Bootloader**

| | |
|---|---|
| bl_boot_eabi.asm | The start-up code used when the Code Composer Studio compiler is being used to build the bootloader. |
| bl_check.c | The code to check whether a firmware update is required or not, or if a firmware update is being requested. |
| bl_check.h | Prototypes for the update check code. |
| bl_commands.h | The list of command and return messages supported by the bootloader. |
| bl_config.h | Bootloader configuration file. This contains all of the possible configuration values. |
| bl_flash.c | The functions for erasing, programming the Flash, and functions for erase and program check |
| bl_flash.h | Prototypes for Flash operations |
| bl_link.cmd | The linker script used when the Code Composer Studio compiler is being used to build the bootloader. |
| bl_main.c | The main control loop of the bootloader. |
| bl_packet.c | The functions for handling the packet processing of commands and responses. |
| bl_packet.h | Prototypes for the packet handling functions. |
| bl_spi.c | The functions for transferring data via the SPI2 port. |
| bl_spi.h | Prototypes for the SPI2 transfer functions. |
| bl_vimram.c | VIM RAM table definition and initialization |
| bw_spi.c | The low-level SPI drive for transferring data |
| bw_spi.h | Prototypes for the low-level SPI2 transfer functions. |
| hw_gio.c | Low-level GIO driver |
| hw_gio.h | Prototypes for low-level GIO driver |
| hw_het.c | Low-level NHET driver |
| hw_het.h | Prototypes for low-level NHET driver |
| hw_interrupt_handler.c | Define the INT handlers |
| hw_pinmux.c | Function for define the pinmux |
| hw_pinmux.h | Prototypes for pinmux functions |
| hw_sci.c | Low-level SCI driver |
| hw_sci.h | Prototypes for low-level SCI driver |
| hw_system.c | Initialize system registers and PLL |
| startup_eabi.c | Global variables initialization |
| sys_intvecs.asm | Interrupt vectors |
| sys_svc.asm | Software INT routines |

## 2 SPI Connections

The sample code provided with the bootloader supports updating via the SPI2 port, which is available on Hercules microcontrollers.

The SPI handling functions are *SPISend()* and *SPIReceive()*. The connections required to use the SPI port are the following four pins: SPI_TX, SPI_RX, SPI_Clk, and SPI_CS. The device communicating with the bootloader is responsible for driving the SPI_RX, SPI_CLK, and SPI_CS pins, while the Hercules MCU drives the SPI_TX pin. The format used for SPI communications is 8-bit data, clock polarity is 0 and clock phase is also set to 0.

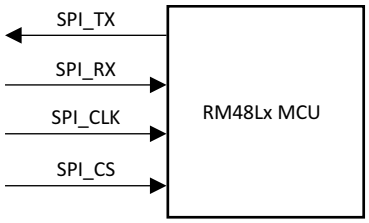


**Figure 1. SPI Connections for RM48x Bootloader**

## 3 Application Update

After HDK reset, the start-up code copies the bootloader from Flash to SRAM, branches to the copy of the bootloader in SRAM, and checks to see if an application update should be performed by calling *CheckForceUpdate()*. If an update is not required, the application is called.

The check for an application update consists of checking the magic word at 0x000017F0 and optionally checking the state of a GIO (GIOA Pin7 in the sample) pin. If either of these tests fail, then the application is assumed to be invalid and an update is forced. The GPIO pin check can be enabled with ENABLE_UPDATE_CHECK in the *bl_config.h* header file, in which case an update can be forced by changing the state of a GPIO pin (with the push button S1 on HDK ). If the application is valid and the GIO pin is not requesting an update, the application is called. Otherwise, an update is started by entering the main loop of the bootloader.

**Table 2. Application Update Options**

| | Invalid | Valid |
|---|---|---|
| Magic word at 0x000017F0 | Other | 0x5A5A5A5A |
| GIO pin | LOW | High |

When performing an update via a SPI port, *ConfigureDevice()* is used to configure the selected SPI port, making it ready to be used to update the firmware. Then, *Updater()* sits in an endless loop accepting commands and updating the firmware when requested. All transmissions from this main routine use the packet handler functions (*SendPacket()*, *ReceivePacket()*, *AckPacket()*, and *NakPacket()*). Once the update is complete, the bootloader can be reset by issuing a reset command to the bootloader.

In the event that the bootloader itself needs to be updated, the bootloader must replace itself in Flash. An update of the bootloader is recognized by performing a download to address 0x0000,0000.

When a request to update the application comes through, the bootloader first erases the sector(s) for application before accepting the binary for the new application. Once all of the application Flash area has been successfully erased, the bootloader proceeds with the download of the new binary. After the application has been successfully programmed to the Flash, the bootloader will write “0x5A5A5A5A” to 0x000017F0.

In order for the bootloader to be separately erasable from the application, the applications must not be placed at the first sector, which is reserved for the bootloader. The default application start address in the sample code is 0x0002 0000.

## 4 Packet Handling

All communications are done via defined packets that are acknowledged (ACK) or not Acknowledged (NAK) by the devices. The packets use the same format for receiving and sending packets. This includes the method used to acknowledge successful or unsuccessful reception of a packet. The basic packet format is shown in Figure 2.

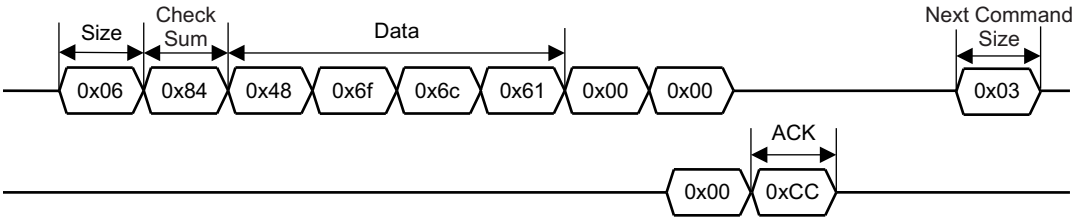


**Figure 2. SPI Packet Format**

The bootloader uses the *SendPacket()* function in order to send a packet of data to another device. This function encapsulates all of the steps necessary to send a valid packet to another device including waiting for the acknowledge or not-acknowledge from the other device. The following steps must be performed to successfully send a packet:

1. Send out the size of the packet that will be sent to the device. The size is always the size of the data + 2.
2. Send out the checksum of the data buffer to help ensure proper transmission of the command. The checksum algorithm is implemented in the *CheckSum()* function provided and is simply a sum of the data bytes.
3. Send out the actual data bytes.
4. Wait for a single byte acknowledgment from the device that it either properly received the data or that it detected an error in the transmission.

Received packets use the same format as sent packets. The bootloader uses the *ReceivePacket()* function in order to receive or wait for a packet from another device. This function does not take care of acknowledging or not-acknowledging the packet to the other device. This allows the contents of the packet to be checked before sending back a response. The following steps must be performed to successfully receive a packet:

1. Wait for non-zero data to be returned from the device. This is important as the device may send zero bytes between a sent and received data packet. The first non-zero byte received will be the size of the packet that is being received.
2. Read the next byte which will be the checksum for the packet.
3. Read the data bytes from the device. There will be packet size - 2 bytes of data sent during the data phase. For example, if the packet size was 3, then there is only 1 byte of data to be received.
4. Calculate the checksum of the data bytes and ensure if it matches the checksum received in the packet.
5. Send an acknowledge or not-acknowledge to the device to indicate the successful or unsuccessful reception of the packet.

The steps necessary to acknowledge reception of a packet are implemented in the *AckPacket()* function. Acknowledge bytes are sent out whenever a packet is successfully received and verified by the bootloader.

A not-acknowledge byte is sent out whenever a sent packet is detected to have an error, usually as a result of a checksum error or just malformed data in the packet. This allows the sender to re-transmit the previous packet.

## 5 Command Definitions

This section defines the list of commands that can be used when communicating with the SPI bootloader. The first byte of the data in a packet should always be one of the defined commands, followed by data or parameters as determined by the command that is sent.

The following two definitions are the values used to indicate successful or unsuccessful transmission of a packet:

```
#define COMMAND_ACK 0x66
#define COMMAND_NAK 0x33
```

The following tables and figures show the details of the commands used in SPI Bootloader.

**Table 3. PING Command Definition for SPI Bootloader**

| | | |
|---|---|---|
| COMMAND_PING | 0x20 | The COMMAND_PING command simply accepts the command and sets the global status to success. The format of the packet is as follows:<br>`Byte[0] = 0x03;`<br>`Byte[1] = checksum(Byte[2]);`<br>`Byte[2] = COMMAND_PING;`<br>The ping command requires the standard 2-byte header followed by the value COMMAND_PING. Since there is only one byte in the command, the checksum is simply equal to COMMAND_PING. Since the ping command has no real return status, the receipt of an ACK can be interpreted as a successful ping to the Flash loader. |

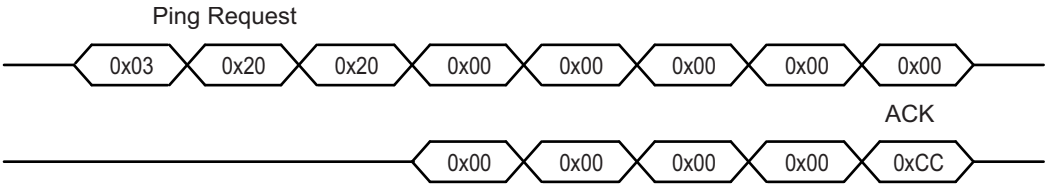


**Figure 3. PING Command for SPI Bootloader**

**Table 4. GET_Status Command Definition for SPI Bootloader**

| | | |
|---|---|---|
| COMMAND_GET_STATUS | 0x23 | This command returns the status of the last command that was issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires one byte in the data of the packet and the bootloader should respond by sending a packet with one byte of data that contains the current status code.<br>The format of the command is as follows:<br>`Byte[0] = 0x03;`<br>`Byte[1] = checksum(Byte[2]);`<br>`Byte[2] = COMMAND_GET_STATUS`<br>The following list shows the definitions for the possible status values that can be returned from the bootloader when this command is sent to the microcontroller.<br>COMMAND_RET_SUCCESS<br>COMMAND_RET_UNKNOWN_CMD<br>COMMAND_RET_INVALID_CMD<br>COMMAND_RET_INVALID_ADD<br>COMMAND_RET_FLASH_FAIL |

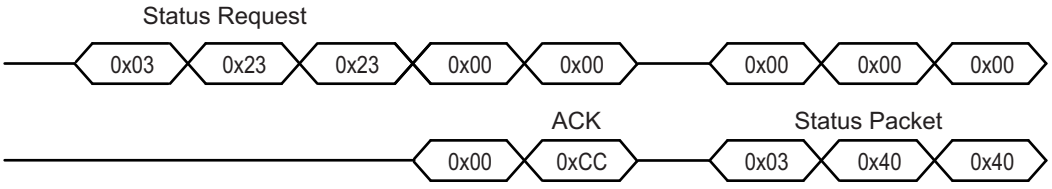


**Figure 4. GET_STATUS Command for SPI Bootloader**

**Table 5. Other Command Definitions for SPI Bootloader**

| | | |
|---|---|---|
| COMMAND_DOWNLOAD | 0x21 | This command is sent to the bootloader to indicate where to store data and how many bytes will be sent by the COMMAND_SEND_DATA commands that follow. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address to start programming data into, while the second is the 32-bit size of the data that will be sent. This command also triggers an erasure of the full application area in the Flash or possibly the entire Flash depending on the address used. This causes the command to take longer to send the ACK/NAK in response to the command. This command should be followed by a COMMAND_GET_STATUS to ensure that the program address and program size were valid for the microcontroller running the bootloader.<br>The format of the command is as follows:<br>`Byte[0] = 11`<br>`Byte[1] = checksum(Bytes[2:10])`<br>`Byte[2] = COMMAND_DOWNLOAD`<br>`Byte[3] = Program Address [31:24]`<br>`Byte[4] = Program Address [23:16]`<br>`Byte[5] = Program Address [15:8]`<br>`Byte[6] = Program Address [7:0]`<br>`Byte[7] = Program Size [31:24]`<br>`Byte[8] = Program Size [23:16]`<br>`Byte[9] = Program Size [15:8]`<br>`Byte[10] = Program Size [7:0]` |
| COMMAND_RUN | 0x22 | This command is sent to the bootloader to transfer execution control to the specified address. The command is followed by a 32-bit value, transferred MSB first, that is the address to which the execution control is transferred.<br>The format of the command is as follows:<br>`Byte[0] = 7`<br>`Byte[1] = checksum(Bytes[2:6])`<br>`Byte[2] = COMMAND_RUN`<br>`Byte[3] = Execute Address[31:24]`<br>`Byte[4] = Execute Address[23:16]`<br>`Byte[5] = Execute Address[15:8]`<br>`Byte[6] = Execute Address[7:0]` |
| COMMAND_GET_STATUS | 0x23 | This command returns the status of the last command that was issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires one byte in the data of the packet and the bootloader should respond by sending a packet with one byte of data that contains the current status code.<br>The format of the command is as follows:<br>`Byte[0] = 0x03;`<br>`Byte[1] = checksum(Byte[2]);`<br>`Byte[2] = COMMAND_GET_STATUS`<br>The following list shows the definitions for the possible status values that can be returned from the bootloader when this command is sent to the microcontroller.<br>COMMAND_RET_SUCCESS<br>COMMAND_RET_UNKNOWN_CMD<br>COMMAND_RET_INVALID_CMD<br>COMMAND_RET_INVALID_ADD<br>COMMAND_RET_FLASH_FAIL |

**Table 5. Other Command Definitions for SPI Bootloader (continued)**

| | | |
|---|---|---|
| COMMAND_SEND_DATA | 0x24 | This command should only follow a COMMAND_DOWNLOAD command or another COMMAND_SEND_DATA command, if more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited by the size of the receive buffer in the bootloader (as configured by the BUFFER_SIZE parameter). The command terminates programming once the number of bytes indicated by the COMMAND_DOWNLOAD command has been received. Each time this function is called, it should be followed by a COMMAND_GET_STATUS command to ensure that the data was successfully programmed into the Flash. If the bootloader sends a NAK to this command, the bootloader will not increment the current address which allows for retransmission of the previous data.<br>The format of the command is as follows:<br>`Byte[0] = 11`<br>`Byte[1] = checksum(Bytes[2:10])`<br>`Byte[2] = COMMAND_SEND_DATA`<br>`Byte[3] = Data[0]`<br>`Byte[4] = Data[1]`<br>`Byte[5] = Data[2]`<br>`Byte[6] = Data[3]`<br>`Byte[7] = Data[4]`<br>`Byte[8] = Data[5]`<br>`Byte[9] = Data[6]`<br>`Byte[10] = Data[7]` |
| COMMAND_RESET | 0x25 | This command is used to tell the bootloader to reset. This is used after downloading a new image to the microcontroller to cause the new application or the new bootloader to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. It can also be used to reset the bootloader if a critical error occurs and the host device wants to restart communication with the bootloader. The bootloader responds with an ACK signal to the host device before actually executing the software reset on the microcontroller running the bootloader. This informs the updater application that the command was received successfully and the part will be reset.<br>The format of the command is as follows:<br>`Byte[0] = 3`<br>`Byte[1] = checksum(Byte[2])`<br>`Byte[2] = COMMAND_RESET` |

## 6 Sample Code

The SPI bootloader Code Composer Studio v5.4 project and Visual C++ host project are available at: http://processors.wiki.ti.com/index.php/Category:RM4.

## 7 References

- *RM48Lx50 16/32-Bit RISC Flash Microcontroller Data Manual* (SPNS174)
- *F021 Flash API Version 2.00.01 Reference Guide* (SPNU501)
- *Aardvark I2C/SPI Adaptor (V5.13) Data Sheet* http://www.totalphase.com/products/aardvark_i2cspi/

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

| **Products** | | **Applications** | |
|---|---|---|---|
| Audio | www.ti.com/audio | Automotive and Transportation | www.ti.com/automotive |
| Amplifiers | amplifier.ti.com | Communications and Telecom | www.ti.com/communications |
| Data Converters | dataconverter.ti.com | Computers and Peripherals | www.ti.com/computers |
| DLP® Products | www.dlp.com | Consumer Electronics | www.ti.com/consumer-apps |
| DSP | dsp.ti.com | Energy and Lighting | www.ti.com/energy |
| Clocks and Timers | www.ti.com/clocks | Industrial | www.ti.com/industrial |
| Interface | interface.ti.com | Medical | www.ti.com/medical |
| Logic | logic.ti.com | Security | www.ti.com/security |
| Power Mgmt | power.ti.com | Space, Avionics and Defense | www.ti.com/space-avionics-defense |
| Microcontrollers | microcontroller.ti.com | Video and Imaging | www.ti.com/video |
| RFID | www.ti-rfid.com | | |
| OMAP Applications Processors | www.ti.com/omap | **TI E2E Community** | e2e.ti.com |
| Wireless Connectivity | www.ti.com/wirelessconnectivity | | |

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265