Application Report
SPNA241–August 2019

CAN Bus Bootloader for Hercules™ Microcontrollers

Qingjun Wang

ABSTRACT

A bootloader enables field updates of application firmware. A controller area network (CAN) bootloader enables firmware updates over the CAN bus. The CAN bootloader described in this application report is based on the Hercules™ ARM® Cortex®-R4/R5 microcontrollers (TMS570LSx, TMS570LCx, RM4x, and RM5x). This application report describes the CAN protocol used in the bootloader and details each supported command.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://git.ti.com/hercules_examples/hercules_examples/trees/master/Bootloaders.

Contents

List of Figures

List of Tables

Trademarks

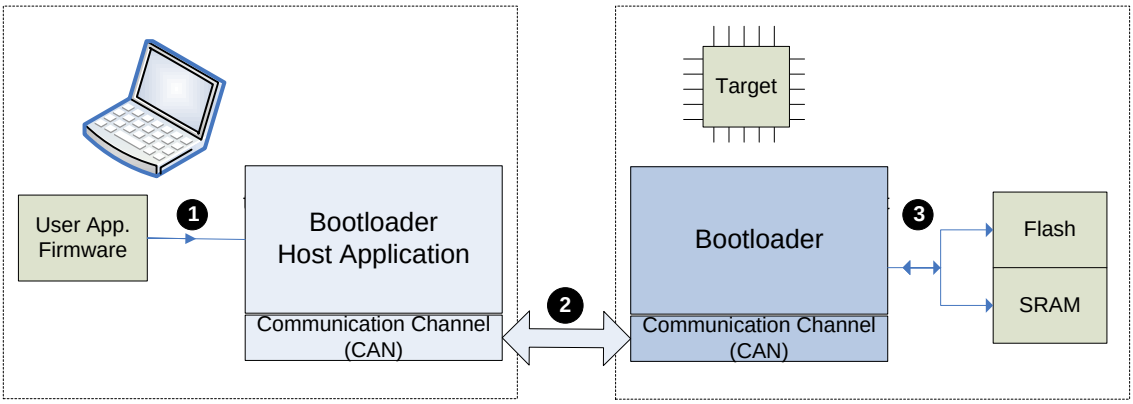
1 Introduction

The CAN bootloader is a small piece of code that resides at the beginning of flash to act as an application loader. It downloads the application through CAN bus, a reliable operation in automotive and industrial control networks. The bootloader also helps designers update the user application for products already deployed in the field.

This document describes how to work with and customize the Hercules CAN bootloader application. Since full source code is provided, the bootloader can be completely customized.

The bootloader on the target device configures the CAN module in communication with PC host through the CAN bus. The bootloader polls the CAN port for messages. After a message is received, the bootloader attempts to decode the incoming commands for flash programming. After the bootloader has successfully downloaded and programmed the whole application image, the bootloader updates the application header. The bootloader jumps to the starting address of the new application image.

The CAN bootloaders for Hercules devices (TMS570LSx, TMS570LCx, RM4x, and RM5x) have been built and validated using Code Composer Studio™ v9 on the Hercules HDKs. The bootloader host application is developed with Visual C++ 2010. Figure 1 and Table 1 show an overview of the source code provided with the bootloader.



(1) Bootloader host application reads the user application.
(2) Bootloader downloads user application to Hercules devices via CAN bus.
(3) Bootloader programs user application into the internal flash of Hercules devices.

Figure 1. Bootloader Process

Table 1. List of Source Code Files Used in CAN Bootloader

bl_main.c	The main control loop of the bootloader
bl_dcan.c	The functions for transferring data via the CAN port
bl_check.c	The code to check if a firmware update is required, or if a firmware update is being requested.
bl_link.cmd	The linker script used when the Code Composer Studio compiler is being used to build the bootloader.
bl_flash.c	The functions for erasing, programming the Flash, and functions for erase and program check
bl_commands.h	The list of commands and return messages supported by the bootloader.
bl_config.h	Bootloader configuration file. This contains all of the possible configuration values.
bl_flash.h	Prototypes for Flash operations
bl_can.h	Prototypes for the CAN transfer functions.
flash_defines.h	Flash memory banks and sectors for Hercules microcontrollers
HALCoGen generated files	Device initialization

2 Hardware Requirements

The hardware required for configuration includes:

- Power supply: 12 V to HDK
- CAN bus: single twisted pair cable with a DB9 socket to connect HDK and NI CAN adaptor.
- Hercules HDK
- NI USB 8473 high-speed CAN adaptor
- A windows 10 PC with Visual C++ 2010

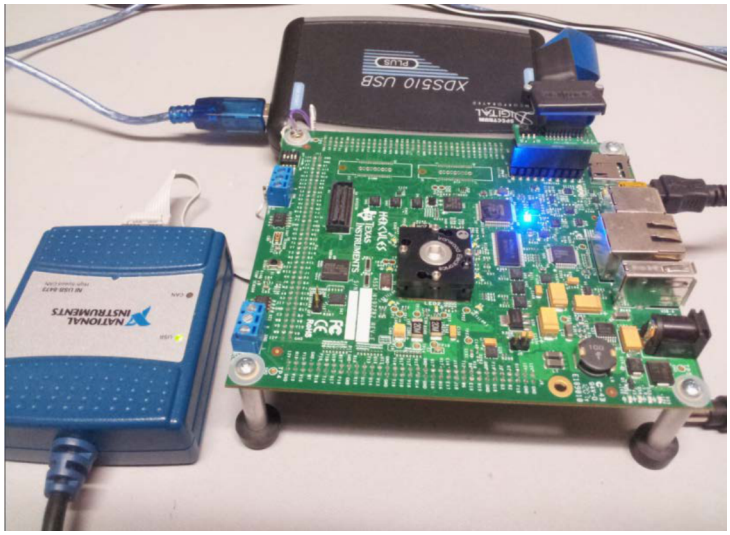


Figure 2. Hardware Setup

3 CAN Settings

The Hercules CAN is compliant with the 2.0A specification with a bitrate up to 1 Mbit/s. It can receive and transmit standard frames with 11-bit identifiers as well as extended frames with 29-bit identifiers. To change the CAN settings for the bootloader, knowledge of the CAN protocol, revision 2.0 is assumed. For details, see the CAN Protocol Revision 2.0 Specification.

Figure 3 shows the essential fields of the standard frame that is used in this CAN bootloader.

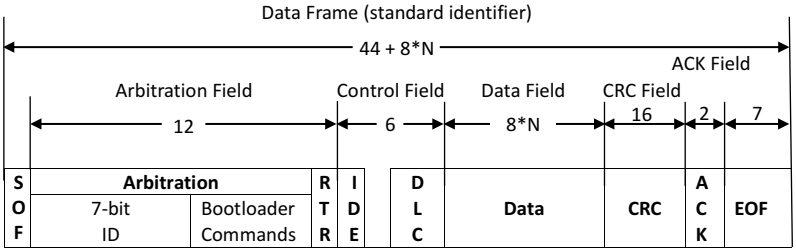


Figure 3. Standard CAN Frame Format

Table 2. Commands Used in Bootloader

Commands	CMD	Description
PING	0x00	See Section 11
GET_ADDR_SIZE	0x01	See Section 11
GET_STATUS	0x03	See Section 11
GET_DATA	0x04	See Section 11
RESET	0x05	See Section 11
ACK	0x06	See Section 11

In this application, the CAN settings are:

- Standard identifier
- Baud Rate: 500 kbps is used by default.
- Functions used: *canInit()*

The transmit settings (from MCU to the host) are:

- Tx mailbox2: On -- `#define MSG_OBJ_BCAST_TX_ID 1` in *bl_dcan.c*
- Tx mailbox1: Off -- `#define MSG_OBJ_BCAST_RX_ID 2` in *bl_dcan.c*
- Tx identifier: 0x5A (device ID) + CMDs (0x00, 0x01, 0x02, 0x04, 0x05, 0x06)
- Functions used: *CANMessageSetTx()*, and *PacketWrite()*

The receive settings (from the host to the MCU) are:

- Synchronization (ACK), 0x06, is in the RX identifier and not in the data field.
- RX identifier: device ID + CMDs
- Error checking: Host re-transmits the frames that have lost arbitration or have been disturbed by errors during transmission.
- Incoming messages can contain from 1 to 8 data bytes.
- Functions used: *CANMessageGetRx()*, *CANMessageSetRx()*, and *PacketRead()*

CAN Bit timing setting:

Two clock domains are provided to the CAN module:

- VCLK: general module clock source
- VCLKA1: clock source to CAN_CLK for generating the CAN Bit Timing (*system.c*)
- Functions used: *canInit()*

Both VCLK and VCLKA can be derived from the same clock source. However, if the frequency modulation in the FMPLL is enabled (spread spectrum clock), then, due to the high precision clocking requirements of the CAN Core, the FMPLL clock source should not be used for VCLKA. Alternatively, a separate clock without any modulation (for example, derived directly from the OSCIN clock) should be used for VCLKA.

Before configuring the CAN module, evaluate your system specifications such as system propagation delay (wire length and transceiver delay), crystal tolerance, and re-synchronization jump width. To initialize the CAN registers in CAN communication, you must define parameters such as baud rate, propagation segment (Prog_Seg), time segment 1 (Phase_Seg1) and time segment 2 (Phase_Seg2). Figure 5 shows the CAN BTR calculations in HalCoGen.

$t_{prop} = 2(t_{bus} + tt_{ransmitter} + t_{receiver})$

t_{bus} = Bus Length (meter) * 5 ns/meter

$t_{transmitter}$ and $t_{receiver}$ can be found in the device-specific transceiver data sheet

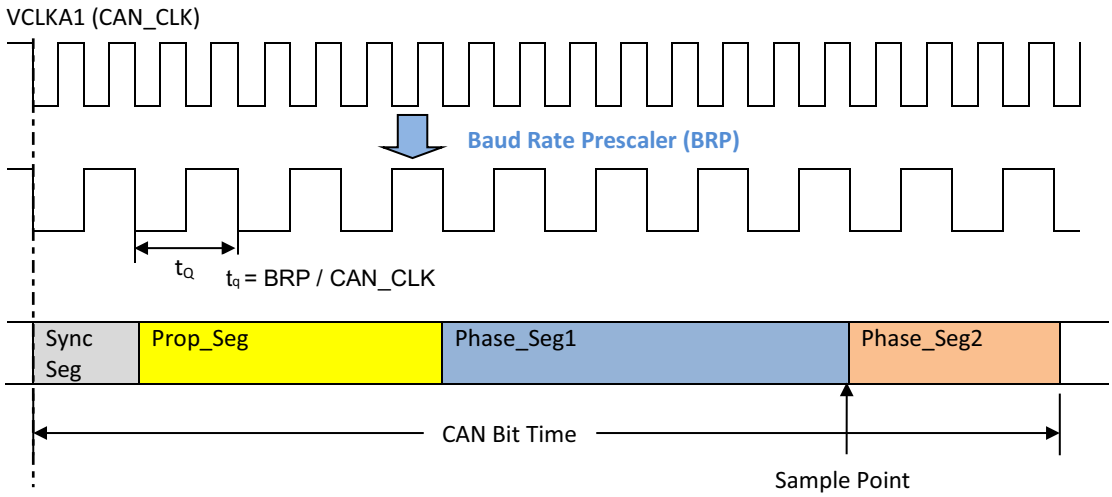


Figure 4. CAN Bit Timing

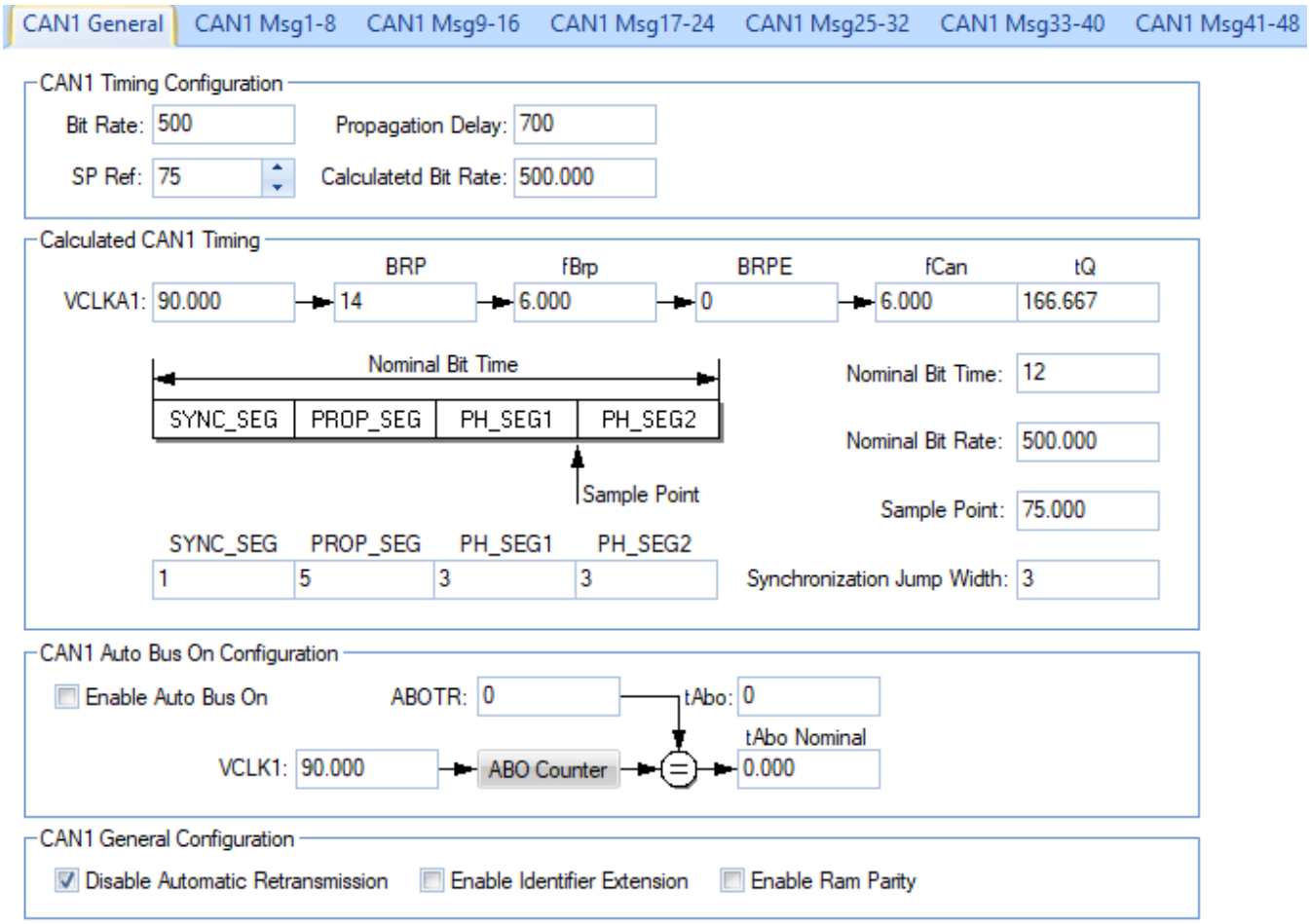


Figure 5. CAN Bit Timing Calculation in HalCoGen

4 Software Coding and Compilation

- The bootloader code is implemented in C. Assembly coding is used only when absolutely necessary. The IDE is TI Code Composer Studio 9.x.
- The bootloader is compiled in the 32-bit ARM mode.
- The compiler is TIV18.12.1.LTS.

5 Exception Vector Table

Exceptions are interruptions of the normal program flow. The Cortex-R4/5 processor usually takes care to preserve the critical parts of the current processor state, so that the normal program flow could be resumed after the exception was handled by the application (saving and restoring of the CPSR and banked Stack Pointers).

The processor state (ARM/Thumb2) and the operating mode can and will change on exception entry. The Cortex-R4/5 processor supports exception entry in ARM and in Thumb2 state, the default after reset as implemented in the Hercules family is the ARM state. The default state is used in this bootloader example.

When the hardware takes an exception, the program counter (PC) is automatically set to the address of the relevant exception vector and the microcontroller begins executing instructions from that address. When the microcontroller comes out of reset, the PC is automatically set to 0x00000000. An undefined instruction sets the PC to 0x00000004, and a data abort sets the PC to 0x00000010, and so forth.

The exception table for ARM Cortex-R devices is held in flash and cannot be modified easily. The reset vector must always point to the start of the bootloader code (0x00000000). One solution is to have an exception service routine (UNDEF, SWI, DABT, PABT) in the bootloader that redirect to the exception vector table located at a fixed location within the application memory space. Using this method added one indirect jump instruction to each exception handler resulting in extra processor load for each interrupt that is processed. Because the content of the IRQ/FIQ interrupt vector registers from the VIM are loaded into PC at 0x18/0x1C of exception vector table, there is no need to redirect IRQ/FIQ.

Another solution is to use the exception handlers in the application. When any exception (UNDEF, SWI, DABT, PABT) occurs, processor fetches the handler address from 0x04/0x08/0x0C/0x10, and jumps to the handler in application. To achieve this, the branch addresses in bootloader exception vector table need to be changed to Application Start Address – 0x8 in ARM state, and Application Start Address – 0x4 in Thumb2 state.

```
;*****************************************************************************
    .sect ".intvecs"
;-----------------------------------------------------------------------------
; import reference for interrupt routines
    .ref _c_int00
;-----------------------------------------------------------------------------
; interrupt vectors
; application start address is 0x10020
        b   _c_int00   ;0x00
        b   #0x10018   ;0x04  UNDEF; application start address - 0x08
        b   #0x10018   ;0x08  SVC  ; application start address - 0x08
        b   #0x10018   ;0x0C  PABT ; application start address - 0x08
        b   #0x10018   ;0x10  DABT ; application start address - 0x08
reservedEntry
        b   reservedEntry     ;0x14
        ldr pc,[pc, #-0x1b0]  ;0x18
        ldr pc,[pc, #-0x1b0]  ;0x1C
```

This bootloader example does not use any interrupt. There is no exception handler for UNDEF, SWI, DABT, and PABT in this bootloader example.

6 ECC Generation for Bootloader Code

The Cortex-R4/R5 CPU may generate speculative fetches to any location within the ATCM memory space. A speculative fetch to a location with invalid ECC, which is subsequently not used, will not create an abort, but will set the ESM flags for a correctable or uncorrectable error. An uncorrectable error will unconditionally cause the nERROR pin to toggle low. Therefore, care must be taken to generate the correct ECC for the entire ATCM space including the holes between sections and any unused or blank flash areas.

The easiest way to achieve this is to use the Linker to generate ECC data rather than the loader. Couple changes should be made:

- Add a ‘vfill = 0xFFFFFFFF' directive to the end of each line that maps to Flash in the Memory{} section of the command file. The 'vfill’ affects only the ECC generation. It instructs the ECC generator to treat the flash as if it were filled with the value 0xFFFFFFFF. It's a virtual fill, because the loader doesn't need to download 4Mbytes.
- Add memory regions corresponding to the ECC area of the flash bank to the Memory{} section.
- Add an ECC {} directive describing the algorithm that matches the device.

Once you make changes to linker command file so that the linker generates ECC for the project, it is necessary to change the loader settings so that the loader doesn't also try to generate ECC. On CCS “Flash Settings”, “Auto ECC Generation” should be unchecked, and “Flash Verification Settings” should be 'None'.

The following is a modified memory map of the linker command file used in the Cortex-R5 CAN bootloader project that you can use to replace the one generated by HALCoGen.

The ECC algorithm directive for Flash ECC devices should be added to generate ECC data.

```
/* Linker Settings                                                    */
--retain="*(.intvecs)"

/* Memory Map  */
MEMORY
{
    /* Add a vfill directive to the end of each line that maps to Flash */
    VECTORS    (X)   : origin=0x00000000 length=0x00000020 vfill = 0xffffffff
    FLASH0     (RX)  : origin=0x00000020 length=0x001FFFE0 vfill = 0xffffffff
    FLASH1     (RX)  : origin=0x00200000 length=0x00200000 vfill = 0xffffffff
    SRAM       (RWX)  : origin=0x08002000 length=0x0002D000
    STACK      (RW)  : origin=0x08000000 length=0x00002000

/* USER CODE BEGIN (3) */
#if 1
    /* Add memory regions corresponding to the ECC area of the flash bank */
    ECC_VEC  (R) : origin=(0xf0400000 + (start(VECTORS) >> 3))
                   length=(size(VECTORS) >> 3)
                   ECC={algorithm=algoL2R5F021, input_range=VECTORS}

    ECC_FLA0 (R) : origin=(0xf0400000 + (start(FLASH0)  >> 3))
                   length=(size(FLASH0)  >> 3)
                   ECC={algorithm=algoL2R5F021, input_range=FLASH0 }

    ECC_FLA1 (R) : origin=(0xf0400000 + (start(FLASH1)  >> 3))
                   length=(size(FLASH1)  >> 3)
                   ECC={algorithm=algoL2R5F021, input_range=FLASH1 }
#endif
/* USER CODE END */
}

/* USER CODE BEGIN (4) */
/* Add an ECC {} directive describing the algorithm that matches the device */
ECC
{
    algoL2R5F021 : address_mask = 0xfffffff8 /*Address Bits 31:3 */
                   hamming_mask = R4   /*Use R4/R5 build in Mask */
                   parity_mask  = 0x0c /*Set which ECC bits are Even & Odd parity */
                   mirroring    = F021  /*RM57Lx and TMS570LCx are build in F021*/
}
/* USER CODE END */
```

7 ECC Generation for Application Code

As stated in Section 6, the ECC values for all of the ATCM program memory space must be programmed into the flash before SECDED is enabled. Before transferring the application image, the address and the image size are transferred first. The bootloader erases the flash sectors based on the address and size, and the ECC area corresponding to those flash sectors. If the image size is not equal to size of one flash sector or multiple sectors, the un-programmed area in the last erased sector contains ECC error.

One way to avoid this ECC error is to fill the un-programmed area by appending 0xFFFFFFFF to the file.

```
MEMORY
{
    VECTORS (X)     : origin=0x00010020 length=0x00000020
    /*sector 4/5 are used for application */
    FLASH_CODE (RX) : origin=0x00010040 length=0x8000 - 0x40  fill=0xFFFFFFFF
    FLASH0  (RX)    : origin=0x00018000 length=0x00200000 - 0x18000
    FLASH1  (RX)    : origin=0x00200000 length=0x00200000
    STACKS  (RW)    : origin=0x08000000 length=0x00001500
    RAM     (RW)    : origin=0x08001500 length=0x0007EB00
}
```

8 During Bootloader Execution

During bootloader execution:

- MCU operates in supervisor mode: The F021 Flash APIs are called in bootloader to erase flash sectors and program the application code. On the ARM Cortex-R4/R5 devices, the flash APIs must be run in a privilege mode (a mode other than user) to allow access to the Flash memory controller registers.
- MCU Clock is reconfigured and is maintained throughout the bootloader execution. The flash can support zero address and data wait states up to a CPU speed of 50 MHz in nonpipelined mode. The flash can support a maximum CPU clock speed in pipelined mode with appropriate address wait state and data wait states. Please make sure the RWAIT is set properly for the specified system frequency.
  - Clock Source: OSCIN = 16 MHz
  - System clock (HCLK): 150 MHz for TMS570LCx and RM57x devices, 160 MHz for TMS570LS31x/12x and RM48/RM46 devices, 100 MHz for TMS570LS07x and RM44 devices, 80 MHz for TMS570LS04x and RM42 devices.
  - Peripheral clock (VCLK): 75 MHz for TMS570LCx and RM57x devices, 90 MHz for TMS570LS31x/12x and RM48/RM46 devices, 100 MHz for TMS570LS07x and RM44 devices, 80 MHz for TMS570LS04x and RM42 devices.
- The interrupt is disabled in bootloader example code. If the interrupt is used in bootloader, it has to be disabled before the code is branched to the application code.
- CAN bit timing: The interrupt is disabled in bootloader example code. If the interrupt is used in bootloader, it has to be disabled before the code is branched to the application code. The default setting is 500 kbps. You can modify the baudrate through HALCoGen.
- F021 Flash API Version 2.01.01 is used. The flash API and its related code must be executed from SRAM.

The user application must be in raw binary format. The hex format is not supported.

9 Bootloader Flow

Figure 6 shows the execution flow of the CAN Bootloader.

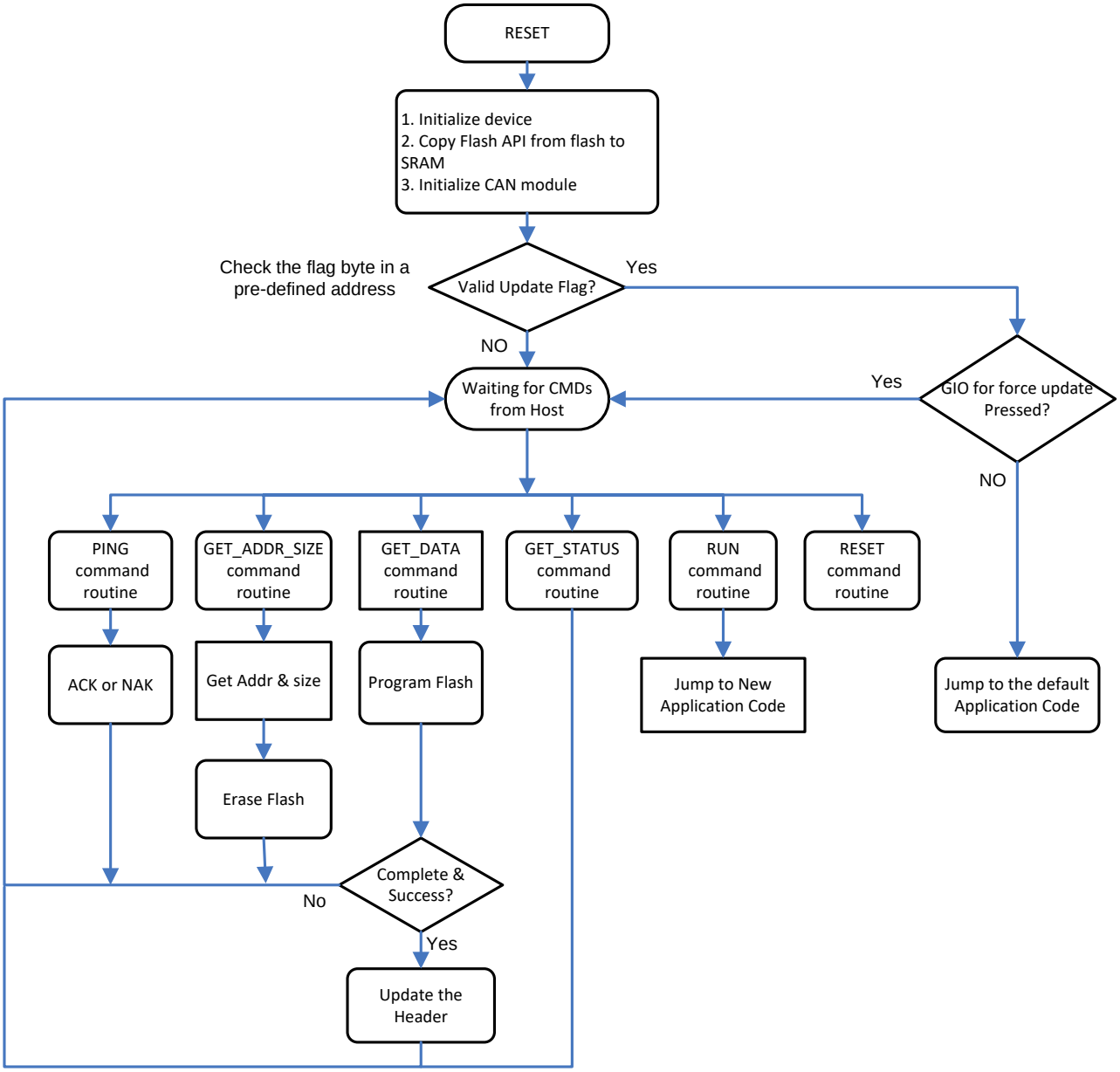


Figure 6. CAN Bootloader Flowchart

10 CAN Bootloader Operation

1. Load the bootloader to Flash.

   The CAN bootloader is built with Code Composer Studio 9.x and loaded through the JTAG port into the lower part of the program Flash memory at 0x00000000.

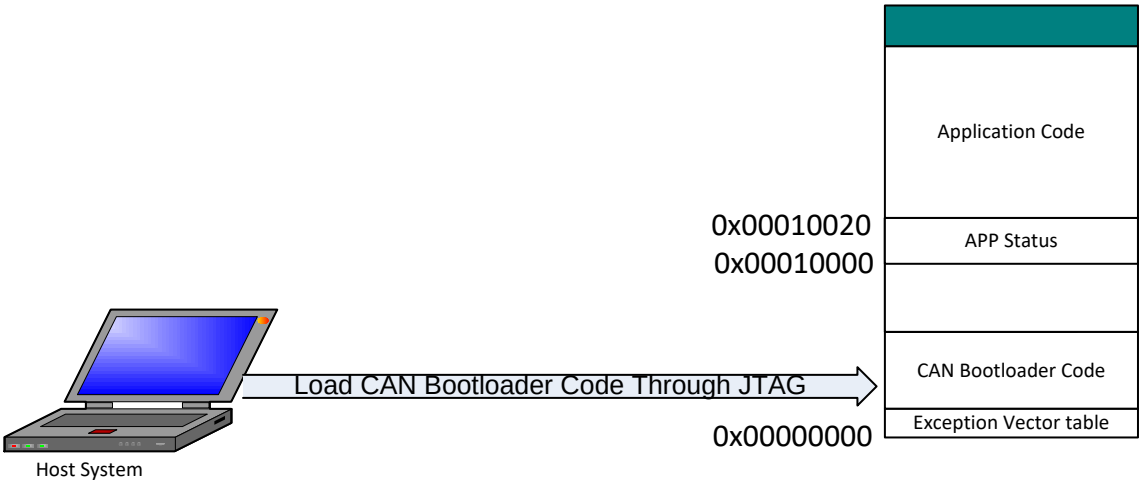


Figure 7. The CAN Bootloader is Loaded Through the JTAG Port

2. Load the user application code.

   After HDK reset, F021 Flash APIs, Flash API related code and variables are copied from Flash to SRAM, and execute the bootloader in Flash.

   First, it checks to see if the GPIO_A7 pin is pulled low by calling *CheckForceUpdate()*. If GPIO-A7 is pulled LOW, the application code is forced to be updated. 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).

   Then, it checks the magic word or flag at 0x00010000. If the flag is a valid number (0x5A5A5A5A), the bootloader jumps to the application code at 0x00010020. If the flag is not the valid number, it configures CAN and SCI, then starts to update the application code by calling *UpdaterCan()*. After all of the application code is programmed successfully, the application header is programmed to 0x10000. The application header consists of three 32bit words that start at 0x10000 in this example. The first word of the header is the application start address, and the second word is the application size, and third word is the status flag. If the application is programmed sucessfully, the flag is 0x5A5A5A5A..

   The CAN bootloader uses Message Box 2 to handle incoming messages; Message Box 1 is used for handling the outgoing messages.

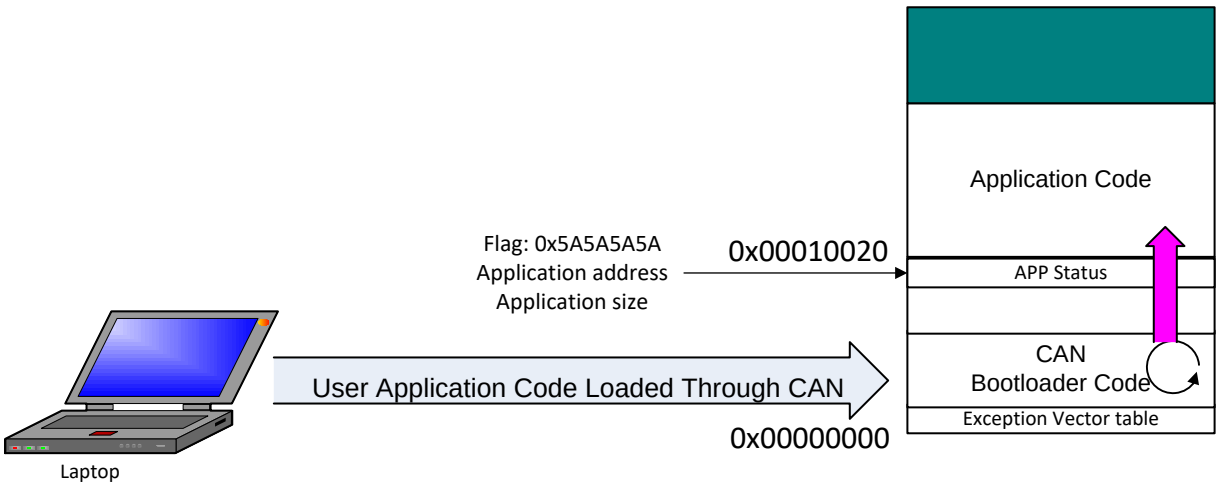


Figure 8. User Application Code is Loaded Through the CAN Bootloader

3. User application is downloaded and programmed. To execute the new application, the bootloader host sends REET command or RUN command to the bootloader.

11 CAN Bootloader Protocol

Messages between a CAN bootloader host and the target use a simple command and acknowledge (ACK) protocol. The host sends a command and within a timeout period the target responds with either an ACK or with a NACK. The command data is combined into message ID. The standard 11 bit message ID is used. Among the 11 bits, the bit 0 to bit 3 is for the bootloader commands, and bit 4 to bit 7 is used for device ID, and the bit 8 to bit 11 is used for manufacturer ID.

The CAN bootloader provides a short list of commands that are used during the firmware update operation. The definitions for these commands are provided in the file *bl_commands.h*. The description of each of these commands is covered in this section.

- CAN_COMMAND_PING (0x00)

  This command is used to receive an acknowledge command from the bootloader indicating that communication has been established. This command has no data. If the device is present, it will respond with a CAN_COMMAND_PING back to the CAN update application.

- CAN_COMMAND_GET_ADDR_SIZE (0x01)

  This command sets the base address for the download as well as the size of the data to write to the device. This command should be followed by a series of CAN_COMMAND_GET_ADDR_SIZE that send the actual image to be programmed to the device. The command consists of two 32-bit values. 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. This Flash erase operation causes the command to take longer to send the CAN_COMMAND_ACK in response to the command, which should be taken into account by the CAN update application.

  The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = Download Address [7:0];
ucData[1] = Download Address [15:8];
ucData[2] = Download Address [23:16];
ucData[3] = Download Address [31:24];
ucData[4] = Download Size [7:0];
ucData[5] = Download Size [15:8];
ucData[6] = Download Size [23:16];
ucData[7] = Download Size [31:24];
```

- CAN_COMMAND_GET_DATA (0x04)

  This command should only follow a CAN_COMMAND_GET_ADDR_SIZE command or another CAN_COMMAND_GET_DATA command when more data is needed.

  Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited to 8 bytes at a time based on the maximum size of an individual CAN transmission. The command terminates programming once the number of bytes indicated by the CAN_COMMAND_GET_ADDR_SIZE command have been received.

  The CAN bootloader sends a CAN_COMMAND_ACK in response to each send data command to allow the CAN update application to throttle the data going to the device and not overrun the bootloader with data.

  This command also triggers the programming of the application area into the Flash. This Flash programming operation causes the command to take longer to send the CAN_COMMAND_ACK in response to the command, which should be taken into account by the CAN update application.

  The LED D7 is flashing until the application update is complete.

  The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = Data[0];
ucData[1] = Data[1];
ucData[2] = Data[2];
ucData[3] = Data[3];
ucData[4] = Data[4];
ucData[5] = Data[5];
ucData[6] = Data[6];
ucData[8] = Data[7];
```

- CAN_COMMAND_RESET (0x05)

  This command is used to tell the CAN bootloader to reset the microcontroller. 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 CAN update application needs to restart communication with the bootloader.

- CAN_COMMAND_REQUEST (0x05)

  This command returns the status of the last command that was issued. This command has no data.

12 Create Application for Use With the Bootloader

In order to upgrade the application using the bootloader, application images are created with a starting address of 0x10020 (default). The bootloader occupies the flash area below this address. To achieve this, the flash start address defined in the linker command file must be changed.

```
/*----------------------------------------------------------------------------*/
/* Linker Settings                                                            */
--retain="*(.intvecs)"

/*----------------------------------------------------------------------------*/
/* Memory Map                                                                 */
MEMORY
{
    VECTORS (X)     : origin=0x00010020 length=0x00000020
    /*sector 4/5 are used for application */
    FLASH_CODE (RX) : origin=0x00010040 length=0x8000 - 0x40  fill=0xFFFFFFFF
    FLASH0  (RX) : origin=0x00018000 length=0x00200000 - 0x18000
    FLASH1  (RX) : origin=0x00200000 length=0x00200000
    STACKS  (RW) : origin=0x08000000 length=0x00001500
    RAM     (RW) : origin=0x08001500 length=0x0007EB00
}

/*----------------------------------------------------------------------------*/
/* Section Configuration                                                      */
SECTIONS
{
    .intvecs : {} > VECTORS
    .text   align(32) : {} > FLASH_CODE
    .const  align(32) : {} > FLASH_CODE
    .cinit  align(32) : {} > FLASH_CODE
    .pinit  align(32) : {} > FLASH_CODE
    .bss     : {} > RAM
    .data    : {} > RAM
    .sysmem  : {} > RAM
}
```

To create an application using TI Code Composer Studio 9.x, use the linker command file included in this application report as a reference for your project. The included linker command file sets up the starting address of exception vector table to 0x10020 for the binary. In the project properties window, type the following command in “Post-Built Steps" to generate binary file:

```
"${CCE_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin" "${BuildArtifactFileName}"
"${BuildArtifactFileBaseName}.bin"
"${CG_TOOL_ROOT}/bin/armofd" "${CG_TOOL_ROOT}/bin/armhex"
"${CCE_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin"
```

The resulting binary will be placed in your project folder, and binary file name is *projectName.bin*.

13 Sample Code for PC-Side Application

The PC-side application is developed using VC++ 2010. The *bl_command.h* defines the commands used for talking with the CAN bootloader on the MCU side. The library and header file for NI-CAN 8473 are included in the project.

The *can_bltest.c* does all the tests for bootlader:

- Opens binary image (user application)
- Sends command to ping MCU bootloader
- Sends starting address and image size to the MCU bootloader
- Sends data of the image to the MCU bootloader
- Sends execution command to run the user application
- Sends Reset command to reset the MCU

The source code of the VC++ project can be downloaded from the following URL: http://git.ti.com/hercules_examples/hercules_examples/trees/master/Bootloaders.

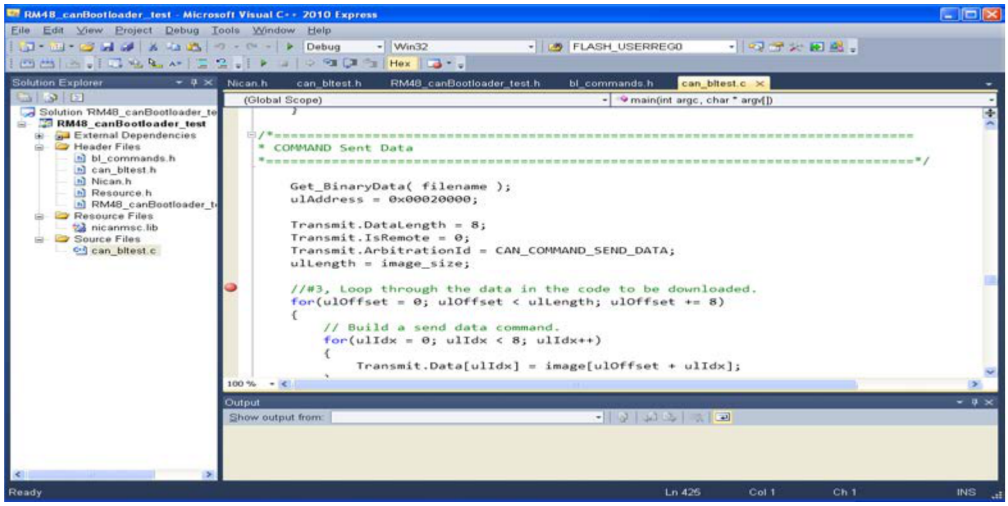


Figure 9. VC++ Project for PC-Side Bootloader

14 References

- Texas Instruments: *TMS570LS0x32 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *TMS570LS04x/03x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *RM42L432 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *RM42x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *TMS570LS1224 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *TMS570LS12x/11x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *RM46Lx50 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *RM46x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *TMS570LS3137 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *TMS570LS31x/21x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *RM48Lx50 16- and 32-Bit RISC Flash Microcontroller Data Sheet*
- Texas Instruments: *RM48x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *TMS570LC4357 Hercules™ Microcontroller Based on the ARM® Cortex®-R Core Data Sheet*
- Texas Instruments: *TMS570LC43x 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Texas Instruments: *RM57L843 Hercules™ Microcontroller Based on the ARM® Cortex®-R Core Data Sheet*
- Texas Instruments: *RM57Lx 16/32-Bit RISC Flash Microcontroller Technical Reference Manual*
- Specification of NI USB-CAN 8473 Adaptor: http://sine.ni.com/nips/cds/view/p/lang/en/nid/203384

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

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