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 note is based on Hercules ARM Cortex-R4 microcontroller. This application note describes the CAN protocol used in the bootloader. It details each supported command.

Table of Contents

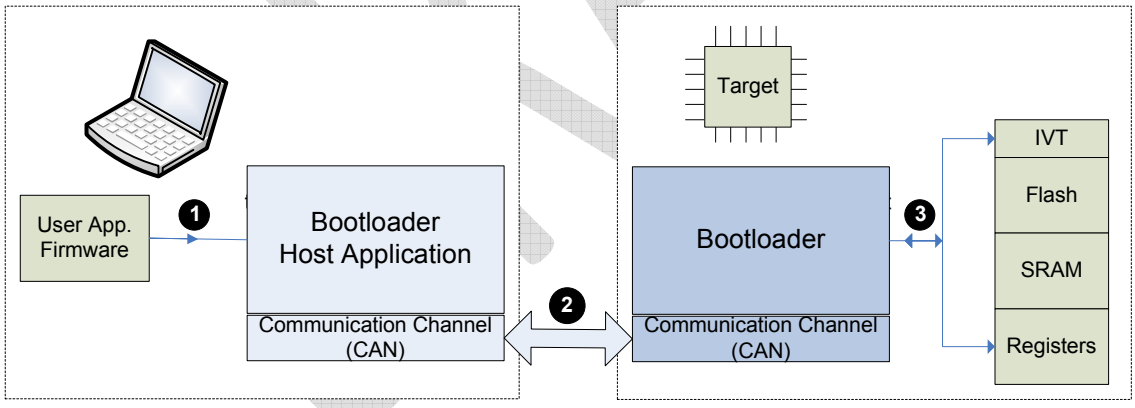
1. Introduction

The CAN bootloader permanently resides in the first flash block of target device. It enables programming of the Hercules microcontroller through its CAN interface. The bootloader also helps designers update the user application code for products already deployed in the field.

This document describes how to work with and customize the Hercules CAN boot loader application. The boot loader is provided as source code which allows any part of the boot loader to 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 internal flash has successfully downloaded the binary image, the bootloader jumps to the starting address of the new application image.

The target side bootloader has been built and validated using CCSv5 on the RM48 Hercules HDK. The bootloader host application which communicates with the target side bootloader is developed with Visual C++ 2010. The following is an overview of the organization of the source code provided with the boot loader.



1. Bootloader host program reads the user application
2. Bootloader transfers user application to target device via communication channel
3. Bootloader target program loads user application into target device memory

Figure 1, Bootloader Process

Table 1, List of Source Code Files Used in CAN Boot Loader

sys_startup.c	The start-up code used when TI's Code Composer Studio (CCS) compiler is being used to build the boot loader.
sys_intvecs.asm	Interrupt vectors
sys_core.asm	Initialize the core registers, stack pointers , memory, etc
system.c	Configure PLL, enable peripherals, etc
bl_main.c	The main control loop of the boot loader.
bl_can.c	The functions for transferring data via the CAN1 port.
bl_check.c	The code to check if a firmware update is required, or if a firmware update is being requested by the user.
hw_pinmux.c	Function for define the pinmux
sci_common.c	Low level SCI driver
bl_link.cmd	The linker script used when the CCS compiler is being used to build the boot loader.
bl_flash.c	The functions for erasing, programming the flash, and functions for erase/program check
bl_commands.h	The list of commands and return messages supported by the boot loader.
bl_config.h	Boot loader 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.
bw_can.h	Prototypes for the low level CAN transfer functions.
hw_pinmux.h	Prototypes for pinmux functions

2. Hardware Requirements

The hardware required for configuration includes:

- Power supply: 12V to HDK
- CAN bus: H, L and GND connecting to CAN1 or CAN2 header on HDK
- Hercules RM48 HDK
- NI USB 8473 high speed CAN adaptor
- PC with windows XP for running VC++ project
- HyperTerminal for message display via RS232 connected to mini USB connector on HDK
    - — Bits/sec: 115200
    - — No parity: none
    - — Stop bit: 1
    - — Flow control: No

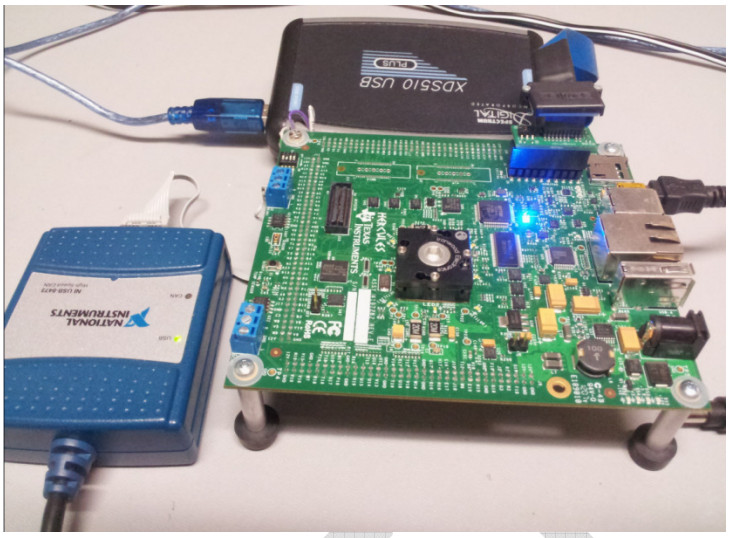


Figure 2, Hardware Setup

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, refer to the CAN Protocol Revision 2.0 Specification. The figure below shows the essential fields of the standard frame which is used in this CAN bootloader.

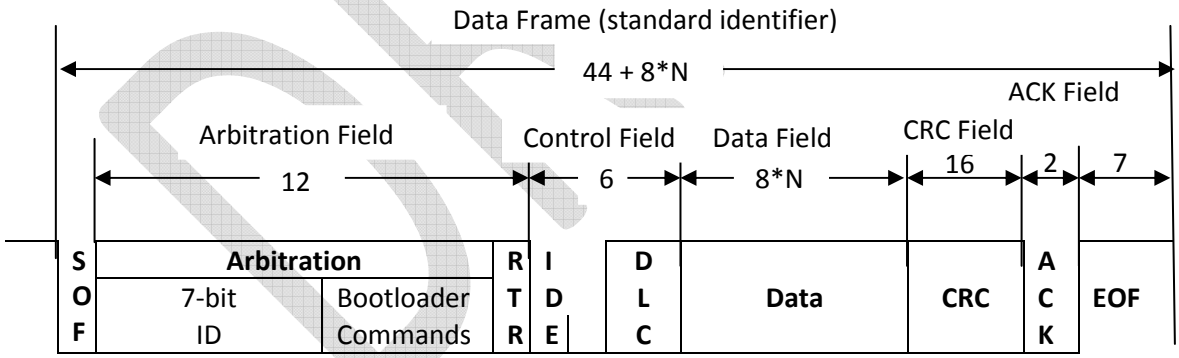


Figure 3, Standard CAN Frame Format

Table 2, Commands Used in Bootloader

COMMANDS	CMD	Description
PING	0x00	See section #9
DOWNLOAD	0x01	See section #9
RUN	0x02	See section #9
GET_STATUS	0x03	See section #9
SEND_DATA	0x04	See section #9
RESET	0x05	See section #9
ACK	0x06	See section #9

In this application, the CAN settings are:

1. Standard identifier (not extended)
2. Bitrate: at the default it is 125 kbps.
3. Functions used: *CANInit()*

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

1. Tx mailbox2: On -- `#define MSG_OBJ_BCAST_TX_ID 1` in *bl_can.c*
2. Tx mailbox1: Off -- `#define MSG_OBJ_BCAST_RX_ID 2` in *bl_can.c*
3. Tx identifier: 0x5A (device ID) + CMDs (0x00, 0x01, 0x02, v03, 0x04, 0x05, 0x06)
4. Functions used: *CANMessageSetTx(), and PacketWrite()*

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

1. Synchronization (ACK), 0x06, is in the RX identifier and not in the data field.
2. RX identifier depends on the commands (0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06).
3. Error checking: Host re-transmits the frames which have lost arbitration or have been disturbed by errors during transmission.
4. Incoming messages can contain from 1 to 8 data bytes.
5. Functions used: *CANMessageGetRx(),CANMessageSetRx(), and PacketRead()*

CAN Bit Timing Setting:

Two clock domains are provided to the CAN module:

1. VCLK: general module clock *(system.c)*
2. VCLKA1: CAN core clock for generating the CAN Bit Timing *(system.c)*
3. Functions used: *CANInit()*

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). Using HalCoGen is a easy way to get the correct BTR value. Figure below shows how CAN BTR calculation looks like in HalCoGen.

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

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

$t_{transmitter}$ and $t_{receiver}$ can be found from transceiver datasheet.

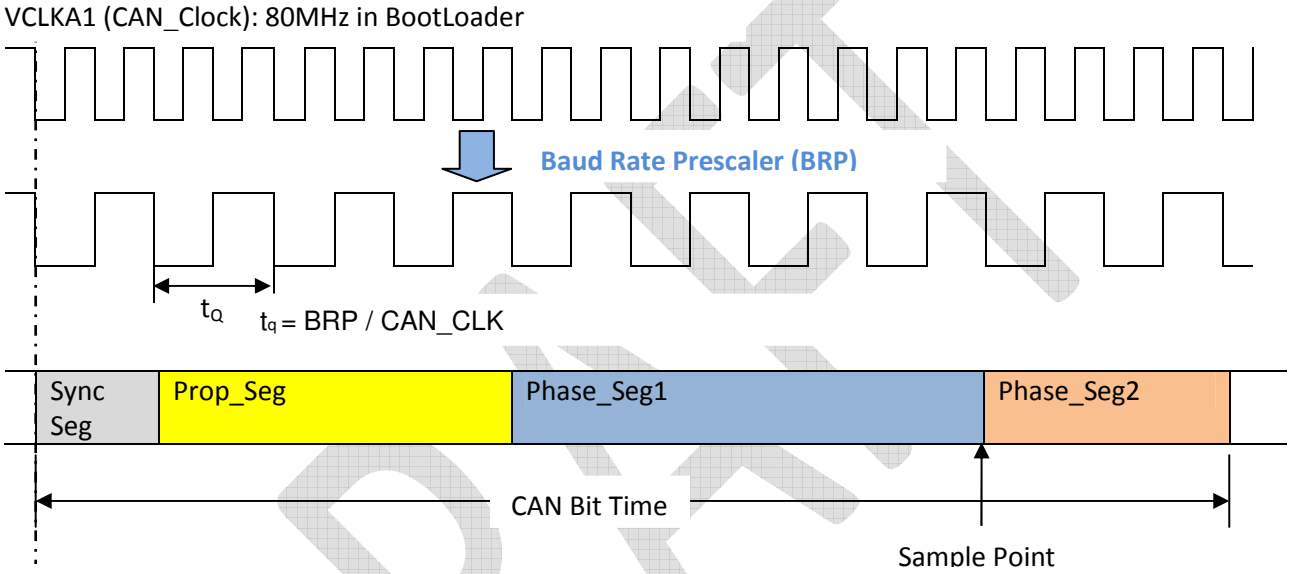


Figure 4, CAN Bit Timing

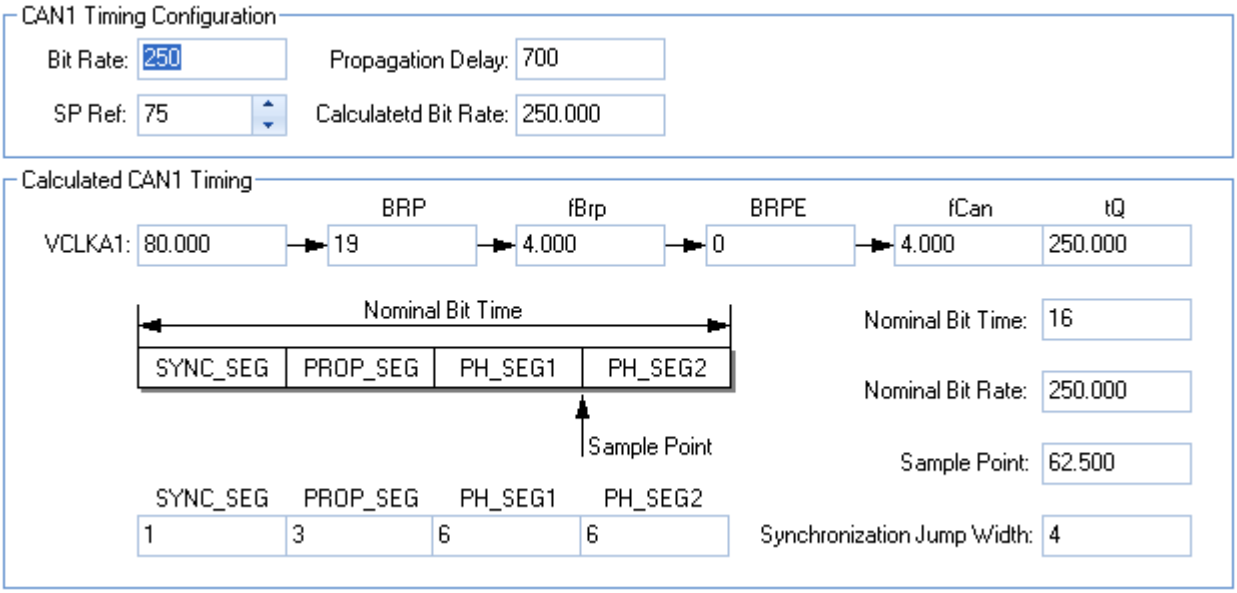


Figure 5, CAN Bit Timing Calculation in HalCoGen

4. Software Coding and Compilation

- The bootloader code is implemented in C, ARM Cortex-R4F assembly coding is used only when absolutely necessary. The IDE is TI CCS5.2.1.
- The bootloader is compiled in the 32-BIT ARM mode.
- The bootloader is compiled and linked with the TI TMS470 code generation tools V 4.9.5.
- The maximum size of the bootloader executable is less than 32KB (Size of the 1st sector flash on RM48 and TMS570LS31x devices).

5. On Reset

On reset, the MCU enters in supervisor mode and starts executing the bootloader. The interrupt vectors are setup as follows:

Table 3, Vector Table in CAN Boot Loader

Offset	Vector	Action
0x00	Reset Vector	Branch to entry point of bootloader (c_int00)
0x04	Undefined Instruction Interrupt	Branch to application vector table
0x08	Software Interrupt	Branch to application vector table
0x0C	Abort (Prefetch) Interrupt	Branch to application vector table
0x10	Abort (Data) Interrupt	Branch to application vector table
0x14	Reserved	Endless loop (branch to itself)
0x18	IRQ Interrupt	Branch to VIM
0x1C	FIQ Interrupt	Branch to VIM

6. During Bootloader Execution

During bootloader execution:

- MCU operates in supervisor mode
- MCU Clock is reconfigured and is maintained throughout the bootloader execution.
  - Clock Source: OSCIN = 16MHz
  - System clock: HCLK = 160Mhz
  - Peripheral clock: VCLK = 80 Mhz
- No interrupts are used.

- CAN bit timing: the basic baud rates such as 125000, 250000, 500000, 750000, and 1000000 are supported. The default setting is 1250000. The baud rate is set in ***bl_config.h***.
- SCI baudrate: The default setting is: 115200:8:N:1. The basic baud rates such as 9600, 19200, 38400, 57600, and 115200 are supported. The baud rate is set in ***bl_config.h***.
- Fix point is used throughout the bootloader execution.
- F021 API V1.5 executes in RAM

Please refer to Technical Reference Manual (SPNU499.pdf) and HalCoGen for device configuration.

7. Bootloader Flow

The following Figure shows the execution flow of the CAN Bootloader.

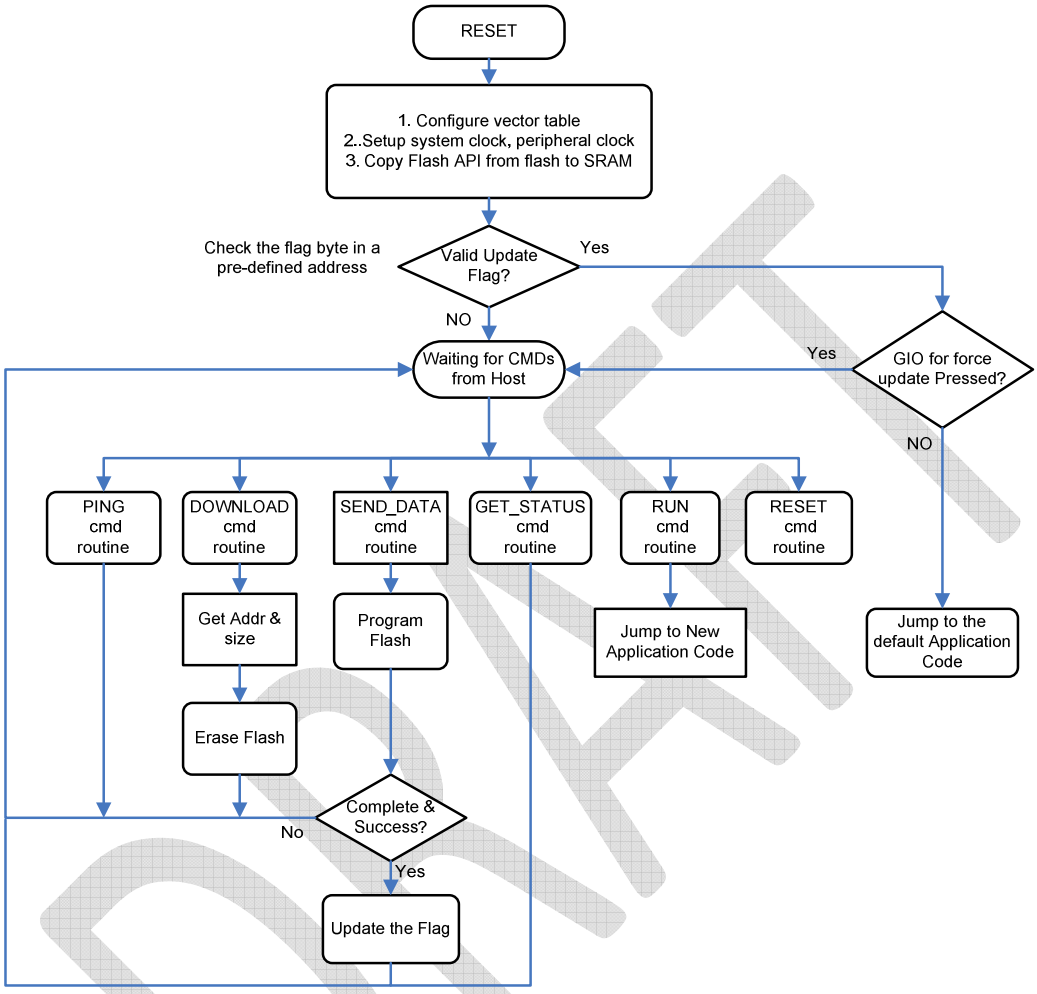


Figure 6, CAN Bootloader FlowChart

8. CAN Bootloader Operation

1. Load the bootloader to flash

The CAN bootloader is built with CCS5.x and loaded through the JTAG port into the lower part of the program memory at 0x0000.

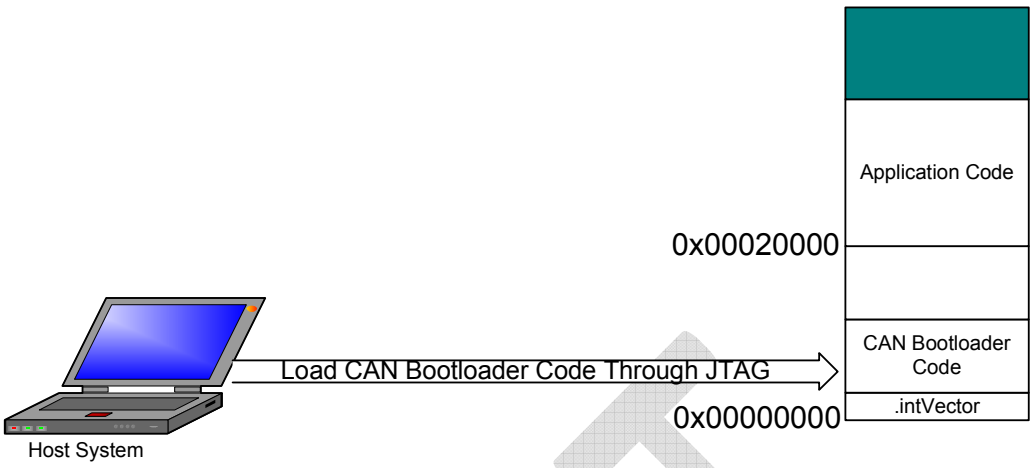


Figure 7, The CAN Bootloader Is Loaded Through The JTAG port

2. Load the user application code.

After HDK reset, the start-up code copies the Flash API of boot loader from flash to SRAM, and execute the boot loader in Flash.

First, it will 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 will check the magic word or flag at 0x0007FF0. If the flag is a valid number (0x5A5A5A5A), the bootloader will jump to the application code at 0x00020000. If the flag is not the valid number, it will configure CAN and SCI, then start to update the application code by calling UpdaterCan(). After all the application code is programmed successfully, the magic work (flag) is also updated to 0x5A5A5A5A.

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

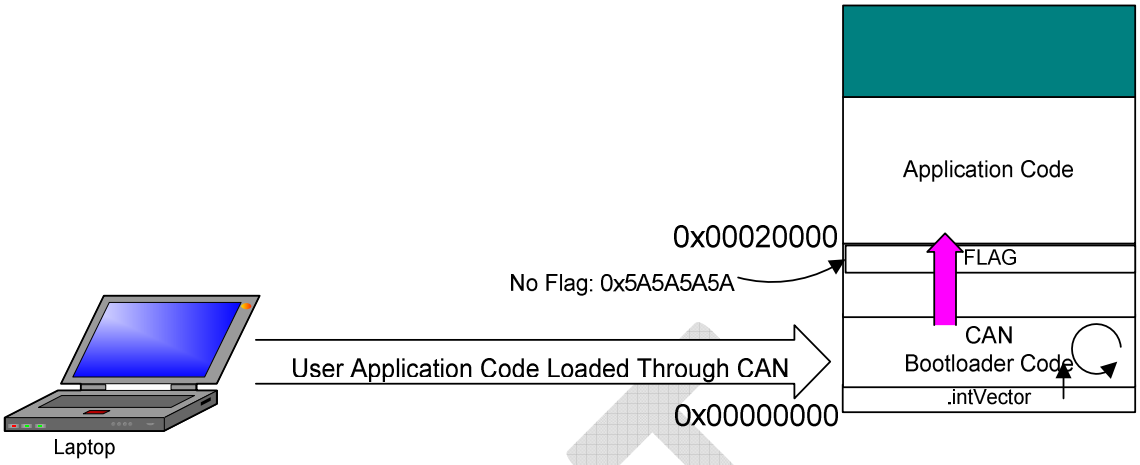


Figure 8, The User Application Code Is Loaded Through The CAN Bootloader.

3. **User application is finally loaded and running after sending the reset command to bootloader.**

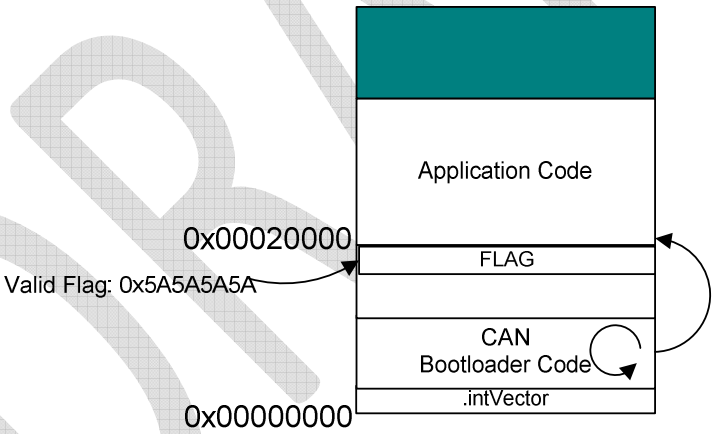


Figure 9, The CAN Bootloader Jumps to Application Code

9. CAN Bootloader Protocol

Messages between a CAN bootloader host and target use a simple command/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 ued 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.

1. **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.

2. **CAN_COMMAND_DOWNLOAD (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_SEND_DATA 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];
```

3. **CAN_COMMAND_SEND_DATA (0x02)**

This command should only follow a CAN_COMMAND_DOWNLOAD command or another CAN_COMMAND_SEND_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_DOWNLOAD command have been received.

The CAN boot loader will send 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 boot loader 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 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];
```

4. **CAN_COMMAND_RESET (0x03)**

This command is used to tell the CAN boot loader to reset the microcontroller. This is used after downloading a new image to the microcontroller to cause the new application or the new boot loader 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 boot loader if a critical error occurs and the CAN update application needs to restart communication with the boot loader.

5. **CAN_COMMAND_REQUEST (0x05)**

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

10. Create Application for Use with the Bootloader

In order to allow future upgrades using the bootloader, applications images must be created with a starting address of 0x20000 (default). The reason for this is that the bootloader itself occupies the flash area below this address. To achieve this, the default flash start address defined in the linker command file must be changed as below:

```
/*----------------------------------------------------------*/
/* Linker Settings                                          */
--retain="*(.intvecs)"
-heap 0x800

/*----------------------------------------------------------*/
/* Memory Map                                               */
MEMORY{
    VECTORS (X)   : origin=0x00020000 length=0x00000020
    FLASH0  (RX)  : origin=0x00020020 length=0x0017FFE0
    STACKS  (RW)  : origin=0x08000000 length=0x00001300
    RAM     (RW)  : origin=0x08001300 length=0x0003ED00
}

/*----------------------------------------------------------*/
/* Section Configuration                                    */
SECTIONS{
    .intvecs : {} > VECTORS
    .text    : {} > FLASH0
    .const   : {} > FLASH0
    .cinit   : {} > FLASH0
    .pinit   : {} > FLASH0
    .bss     : {} > RAM
    .data    : {} > RAM
    .sysmem  : {} > RAM
}
/*----------------------------------------------------------*/
```

Figure 10, The Linker file for Application

To create an application using TI CCS5.x, use the linker files included with this application note for your project. The included linker files set up the starting address of Vector Table and Memory Regions to 0x20000 for the binary. In the project properties window, type the following command in “Post-Built Steps Command”:

```
"${CCE_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin.bat"
"${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"
"${CG_TOOL_ROOT}/bin/ofd470.exe"
"${CG_TOOL_ROOT}/bin/hex470.exe"
"${CCE_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin.exe"
```

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

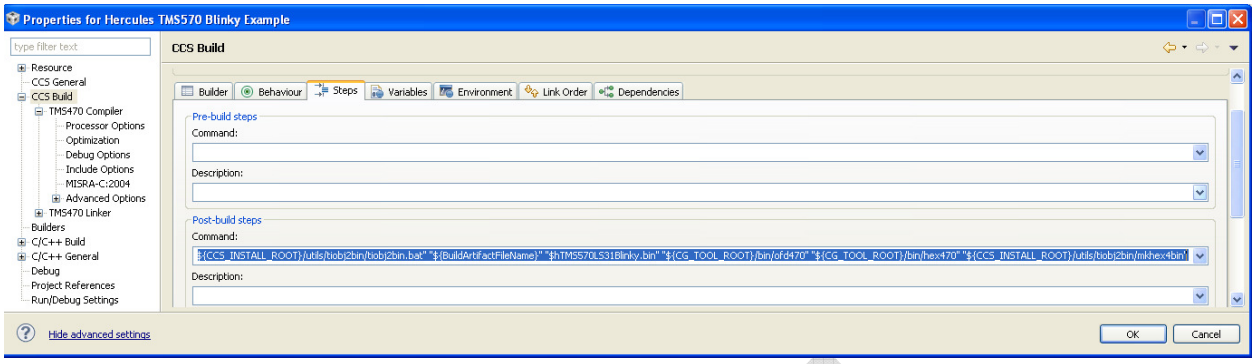


Figure 11, Setup Project Property to Generate Binary File for Bootloader

11. 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 CAN bootloader on 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:

1. Open binary image (user application)
2. Send command to ping MCU bootloader
3. Send starting address and image size to MCU bootloader
4. Send data of the image to MCU bootloader
5. Send execution command to run the user application
6. Send Reset command to reset the MCU

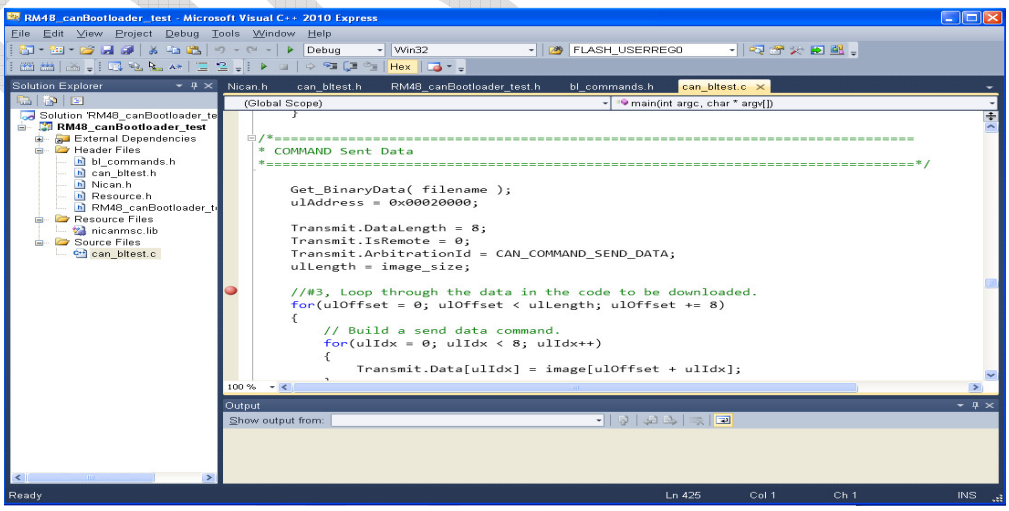


Figure 12, VC++ Project for PC Side Bootloader

12. References

- Datasheet of RM48Lx50 16/32-Bit RISC Flash Microcontroller (SPNS174)
- F021 Flash API (V1.5, SPNU501A)
- Specification of NI USN-CAN 8473 Adaptor