Chapter 1

The ARM_Foundation Documentation

1.1 Bootloader Functionality Overview

The ULC Robotics Bootloader is a program that is flashed onto microcontrollers in ULC Robotics products; it is run on power-up / reset before the application code to allow the operator to diagnose issues and potentially upload new application code. Any application code loaded onto ULC Robotics devices should also be able to jump directly to the bootloader at anytime via a CAN OD Entry Program Control (0x1F51).

After hardware and software have been initialized the STM32 RCC_CSR Reset flags are checked to determine the cause for the Bootloader Execution. No matter the cause for the reset a CAN Emergency Message will be transmitted: ERRCODE_RESET (0x6001) with a status by indicating cause reset_errot_t.

- Power-Up Reset
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_NONE (0x00)
  - Enter EXECMODE_POWER
- Watchdog Reset
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_COP (0x01)
  - Enter EXECMODE_RESET
- Intentional NVIC Software Reset
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_SOFT (0x02)
  - Enter EXECMODE_RESET
- NRST Pin Reset:
  - *NRST Pin is issued by virtually all resets, This reset is only used if no other flags also apply*
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_PIN (0x03)
  - Enter EXECMODE_RESET
- Option Byte Load Reset
  - *When Flash Protection bits are altered they must be loaded via an Option Byte Load Reset*
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_OPTBYTE (0x04)
  - Enter EXECMODE_RESET
- No Reset Flags Detected, Assume Direct Jump:
  - Send CAN Emergency ERRCODE_RESET (0x6001), RESET_ERROR_JUMP (0x05)

- Enter EXECMODE_RESET

Note: On all but Power-Up Resets the contents of RAM are not zeroed. This feature can be used by the application to indicate alternate reasons for reset caused by Cortex M4 Bus, Hard Fault, or Usage Fault Exceptions (Stack-overflow etc). This feature is currently un-utilized but implemented.

When sending an EMCY Frame this additional byte is sent along with the Reset Status Type. A series of incrementing values are placed into this ram storage while attempting to boot the application. 0x00 is placed into this RAM location when initial Application Jump is requested; if the Application Jump fails during the Bootloader portion this value will be sent with the Emergency Value. 0x01 is placed into RAM if the Application startup Fails during the Application portion. 0x02 is placed into RAM if Application start-up succeeds. Further values are reserved for the future:Stack-Overflow etc.

If a Reset of Reset_status 'Intentional NVIC Software Reset' is received after attempting to start the Application, the App_byte should be checked to determine the cause.

CAN Reset Emergency:

Error Code (hex)	Error Name	Additional Value Meaning	Additional Data Length (bytes)
0x6001	Device Reset Error	Reset_status + App_byte	2

Descriptions of Execution Modes:

- EXECMODE_POWERUP - The only mode that will cause the Bootloader to automatically run the application, and only if both the Bootloader and application CRC32's are valid.
- EXECMODE_NORMAL - The Bootloader will not automatically enter application. It will check CRC32s, send a CAN Boot message and wait for further instructions. In this execution mode the flash can be erased and programmed.
- EXECMODE_TIMER - Bootloader was in EXECMODE_POWERUP implying both CRC32's are good. Now wait for WAIT_TIMEOUT (5) seconds before automatically entering application to give operator application time to intervene.
- EXECMODE_RUN - Attempt to run the application; either because EXECMODE_TIMER has expired or we were directed to via an SDO or NMT Command. Check application CRC32 before execution. If CRC32 is valid reset leaving a command to run the application as described in STM32 Boot Process, else check if Application code is empty and send appropriate messages.
- EXECMODE_RESET - Not powerup reset so caused by either Watchdog, Direct Jump, NVIC_Sys call or Debug. Check CRCs but immediately go to EXECMODE_NORMAL even if CRC32's are fine.

After the initial Execution mode is determined the Bootloader checks the CRC32 of itself (details of the Bootloader Size, Location, and CRC are located in STM32 Boot Process). If it determines the Bootloader CRC to be valid it will take not action, conditionally allowing the Bootloader to continue into EXECMODE_TIMER. If it determines the Bootloader CRC to be invalid it will send CAN Emergency ERRCODE_INTERNAL (0x6100) with the calculated checksum and enter EXECMODE_NORMAL.

The Bootloader then checks the CRC32 of the Application code. If it determines the Application CRC to be valid it will take not action, conditionally allowing the Bootloader to continue into EXECMODE_TIMER. If it determines the Bootloader CRC to be invalid it will check to see whether the Application Area is empty. If the Application area is empty it will send CAN Emergency ERRCODE_USER_EMPTY (0x6201), 0x01, otherwise it will revert to sending CAN Emergency ERRCODE_USER (0x6200) with the calculated checksum; in either case it enters EXECMODE_NORMAL.

Note: Due to the significantly larger sizes of Flash pages on some STM32 devices, their erasure can be slow. The original ESA Firmware erased pages as needed which could cause excessive waiting and SDO timeouts. To accomodate larger flash sizes, a new method of programming was devised; it is described in Bootloader State Machine Detail. Generally, if a device's Application Code has an invalid CRC32 the whole application section must be erased via a CAN SDO command and the device will reset when finished and upon reset indicate an empty Application area with CAN Emergency messages.

Finally the bootloader will check the current Execution Mode to determine whether to run the application or send a CAN boot up message:

- If the Reset Flags indicated a Power-Up Reset and the Bootloader and Application CRC32 are valid the Execution Mode will still be EXECMODE_POWERUP; a 60 second timer will be initiated (to allow the operator application to intervene before the application starts) and the Bootloader will enter EXECMODE_TIMER.
- Otherwise, the Reset Flags indicated a Reset other than Power-Up Reset and/or one or more of the C↩ RC32's were Invalid; in these cases a CAN boot-up message will be sent and the Execution mode set to EXECMODE_NORMAL; the Bootloader will await further instructions.

In the Bootloader main loop the Bootloader should only be in one of three states : EXECMODE_NORMAL, EXE↩ CMODE_TIMER, or EXECMODE_RUN.

- If in EXECMODE_NORMAL the Bootloader is waiting for further instructions (such as programming the flash). Once the flash has been programmed the Bootloader can be changed to EXECMODE_RUN by sendin 0x01 to the Program Control OD Entry (0x1F51, 0x01).
- If in EXECMODE_TIMER the Bootloader is waiting to run the application. It will wait for WAIT_TIMEOUT (5) seconds then change to EXECMODE_RUN and attempt to run the application.
  - During this period, if NMT Reset Node or Communications messages are received addressed to all nodes or this node ID EXECMODE_RUN will immediately be entered.
  - During this period, if an SDO command is received, EXECMODE_NORMAL will immediately be entered and automatic running of the application cancelled.
- If in EXECMODE_RUN the Bootloader will check the CRC32 of the application and if valid, reset itself leaving a command to run the application as described in STM32 Boot Process.

1.2 Project Specific Changes

The bootloader is designed to operate the same for all ULC Robotics projects, however, differences in microcontroller hardware and requirements of the specific hardware connected to the microcontroller's pins requires that the firmware be altered and re-compiled for each project. These changes include:

- Changes to bootloader CRC check and Flash loading to accomodate different flash sizes.
- Differing implementation of init_hw_main_board_defaults()
  - Each Project will likely have differing requirements for default levels of each microcontroller pin to ensure safe operation.
    - init_hw_main_board_defaults_and_get_addr() is called during initialization so that all pins remain in a safe state while the bootloader is running without relying on having valid application code loaded onto the microcontroller
- Changes in configuration of Clock depending on Oscillator / Crystal used in project

1.3 Bootloader State Machine Detail

1.4 CAN Entries and Emergencies Overview

1.4.1 Object Dictionary Reference

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
0x1000	0x00	**Device Type**	uint32_t	4	RO	0x74F6↩ F62	"boot"
0x1001	0x00	**Error Register**	uint8_t	1	RO	0x00	
0x1003		**Predefined Error**					
	0x00	# of Entries	uint8_t	1	RW	0x01	0
	0x01	Error Field 1	uint32_t	4	RO		
	...	...	...	...	...		
	0x0A	Error Field 10	uint32_t	4	RO		
0x1018		**Identity Object**					
	0x00	# of Entries	uint8_t	1	RO	0x04	
	0x01	Vendor ID	uint32_t	4	RO		
	0x02	Product Code	uint32_t	4	RO		
	0x03	Firmware Revision	uint32_t	4	RO		
	0x04	Serial Number	uint32_t	4	RO		
0x1F50		**Download Program Data**					
	0x00	# of Entries	uint8_t	1	RO	0x01	
	0x01	Program Number 1	uint32_t	4	RW		
0x1F51		**Program Control**					
	0x00	# of Entries	uint8_t	1	RO	0x01	
	0x01	Program Number 1	uint8_t	1	RW	0	App Not running
0x2FFF		**Special Functions**					
	0x00	# of Entries	uint8_t	1	RO	0x02	
	0x01	Lock Bootloader	uint32_t	4	RW	0x656↩ E6F6E	"none"
	0x02	Erase Application	uint32_t	4	WO		

1.4.2 CAN Emergency Error Codes

Error Code (hex)	Error Name	Additional Value Meaning	Additional Data Length (bytes)	Additional Data Value
0x0000	No Error / Reset		0	
0x1000	Generic Error		0	
0x2000	Generic Current Error		0	
0x3000	Generic Voltage Error			
0x4000	Generic Temperature Error			
0x5000	Generic Device Software Error			
0x6000	Generic Device Hardware Error			
0x6001	Device Reset Error	Reset_status + App_byte	2	
		Reset No Error + App_byte	2	0x00
		Reset Watchdog + App_byte	2	0x01
		Reset Software + App_byte	2	0x02
		Reset Pin + App_↩ byte	2	0x03
		Reset Opt Byte + App_byte	2	0x04
		Reset Direct Jump + App_byte	2	0x05
0x6100	Bootloader CRC Error	Calc'd CRC	4	
0x6200	User Software CRC Error	Calc'd CRC	4	
0x6201	User Software Flash Empty		0	
0x7000	Generic Device Profile Error			
0x8000	Generic Monitoring Error			
0x9000	Generic External Error			

1.4.3 Details of CAN implementation and Entries

1.4.3.1 Error Register and Error List

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
0x1001	0x00	**Error Register**	uint8_t	1	RO	0x00	
0x1003		**Predefined Error**					

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
	0x00	# of Entries	uint8_t	1	RW	0x01	0
	0x01	Error Field 1	uint32_t	4	RO		
	...	...	...	...	...		
	0x0A	Error Field 10	uint32_t	4	RO		

Bit 0 in the Error register is set if atleast one error occurred and was added to the error list. Resetting the error list also resets the error register.

The Predefined Error is a dynamic list, its number of sub-indicies can change. Subindex 0 always has the current number of entries in the list and indicates the highest sub-index that can be read. After a reset, and if there is no error, subindex 0 is 0x00 and the list is empty. The latest error is always added at the top (lowest sub-index) and the highest available sub-index has the oldest error.

Error Field Bit Mapping:

31 - 16	15 - 0
Additional Information	Error Code

Bits 0 - 15 Error Code (hex)	Meaning	Bits 15 - 31 Additional Info
0x6100	Internal Software Error: Bootloader CRC	2 LSB Bytes of calc'd CRC
0x6200	User Software Error: Application CRC	2 LSB Bytes of calc'd CRC

Note: Writing 0x00 to entry [0x1003, 0] clears the Predefined Error Fields and Error Register Status

1.4.3.2 Download Program Data

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
0x1F50		**Download Program Data**					
	0x00	# of Entries	uint8_t	1	RO	0x01	
	0x01	Program Number 1	uint32_t	4	RW		

Download Data File Format

The download data is a binary Intel hex file format. Format variant is little-endian 32-bit INTEL32 (I32HEX) file format. Due to the conversion into binary, the leading colon character (":") as well as all end-of-line characters are omitted. Therefore, the file is a straight set of records of the following type:

Byte #	Description
1	Length (n)
2	Address High
3	Address Low
4	Record Type
5	Data Byte 1

Byte #	Description
...	...
i-1	Data Byte n
i	Checksum
i+i	Length of Next Record

This format allows very straightforward conversion from the output of common Development tools. The allowed record types are:

Hex Code	Record Type
0x00	Data
0x01	End-Of-File
0x04	Extended Linear Address
0x05	Start Linear Address

In order to successfully run the loaded application, a 32-bit application CRC32, located at the end of the flash area, has to be appended to the load file. The CRC32 checksum uses the `STM32 CRC Peripheral`.

Default Settings:

- Polnomial: 0x104C11DB7
- Initial Value: 0xFFFFFFFF
- Input: Word (4-byte)
- Input Inversion: Disabled
- Output Inversion: Disabled
- Include CRC32 memory in CRC Calculation?: No

The same CRC32 method is used for the bootloader flash area and the bootloader checksum.

> Note: Due to the use of CRC32 and I32HEX, the commandline tool provided by Embedded Systems Academy (hexsum.exe) will not work. A new tool must be created.

> Note: Because the I32HEX format used on the STM32 Bootloader only changes the interpretation of the bytes received, the ESA Provided "COLOADERPRO.EXE" can still be used to upload the I32HEX files.

1.4.3.3 Program Control

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
0x1F51		**Program Control**					
	0x00	# of Entries	uint8_t	1	RO	0x01	
	0x01	Program Number 1	uint8_t	1	RW	0	App Not running

The bootloader implements the Program Control (0x1F51) as per DS301.

Sub-index 1 indicates the state of the loaded application. In the bootloader this value is 0x00 as the Application is not running. In the Application this value is 0x01 as the Application is running. To change between Bootloader execution and Application execution, write the desired execution to 0x1F51, 1: 0x00 = Bootloader, 0x01 = Application.

Note: Starting the Application From the Bootloader can be triggered via the NMT Reset Commands.

1.4.3.4 Special Functions

Index (hex)	Sub-Index (hex)	Name	Data Type	Length (bytes)	Access	Default	Meaning
0x2FFF		**Special Functions**					
	0x00	# of Entries	uint8_t	1	RO	0x02	
	0x01	Lock Bootloader	uint32_t	4	RW	0x656E6↩F6E	"none"
	0x02	Erase Application	uint32_t	4	WO		

The Bootloader implements a manufacturer-specific object for two special functions:

- Locking Bootloader (0x2FFF, 0x01)
- Erasing the Application Area (0x2FFFm 0x02)

To perform these actions a special code must be written to the respective OD Entries.

Locking the Bootloader can be accomplished by writing the password "prot" (0x746F7270) to (0x2FFF, 0x01). It will enable Flash protection on the Bootloader Area preventing overwriting of the Bootloader area by JTAG without a complete Chip Erase. If this OD Entry is read it will indicate the protection state of the Bootloader: "prot" if the Bootloader protection is active and "none" if it is not.

Erasing the Application Area can be accomplished by writing the password "eras" (0x73617265) to (0x2FFF, 0x02). It will erase the Application Area and reset the microcontroller so that it can reboot and indicate its erased status.

Writing anything other than their respective passwords to these entries will cause an SDO Abort.

1.5 Adding New Projects to the Bootloader

The Bootloader is a single code-base that should be the same for all projects at ULC Robotics; however, given that each project may use a different MCU (hopefully in the STM32F3 Series), the various pieces of functionality may need to be mapped to different peripherals or pins on different devices.

Complicating this requirement is the desire to have the Bootloader build for both the production boards and development boards.

Requirements:

- Single general bootloader initialization file .c for each series that utilizes project specific mappings via C Preprocessor Macro Defines to initialize all peripherals and pins.
- Single Project specific initialization .c file for each project that utilizes project specific mappings via C Preprocessor Macro Defiens to be able to default all GPIOs to safe settings.
- Single C Preprocessor Define File per Board (one for Dev Board, one for Production Board.

Description of files:

- 'src/device/stm32f3xx/projects' - Folder that contains all Project Specific settings.
- 'src/device/stom32f3xx/projects/proj_name' - Folder that contains a given Project's specific settings.
- 'hw_proj_name.c - Project specific initialization file
  - Implements GPIO configuration for pins that aren't used in Bootloader but need to be set to specific setting for Safety
- 'dev_boart_1.h' - C Preprocessor Macro Define mappings for Development board of a given project.
- 'prod_board_1.h' - C Preprocessor Macro Define mappings for Production board of a given project.

1.5.1 Adding New Projects to the Bootloader

1.6 STM32 Boot Process

On RESET, both Power and Software, the STM32F3 hardware will begin execution from 0x00000000; this address is a remap of either Main Flash, System Flash, or SRAM depending on the setting of BOOTx pins. Once the program at that remapped location is run, it can remap the Vector Table using Cortex M4's System Control Block (SCB) Vector Table Offset Register (VTOR). Subsequently any interrupts will be grabbed from the offset position until RESET.

For the reasons above the Bootloader is placed in Main Flash (0x08000000) so that the Bootloader is ALWAYS run after RESET. The Application is placed in Main Flash at a 16k offset (0x08004000).

The Bootloader can run application by placing magic number into SRAM; when it resets, before initializing any peripherals it will check SRAM and if it sees magic number it will instead launch application via a call to the application's Reset_Handler (0x08040004) which will initialize and prepare the microcontroller

Element 4 STM32F303 Variant:

- Bootloader:
- Start : 0x08000000
- End : 0x08003FFF
- Length: 16k = 0x3FFF
- Checksum Addr: 0x08003FFC

Application:

- Start : 0x08004000

- End : 0x0804FFFF
- Length: 240k = 0x4BFFF
- Checksum Addr: 0x0804FFFC

Boot Process:

- Reset_Handler is called which calls _start()
- _start:
  1. initializes hardware via a call to _initialize_hardware_early
  2. Call user_pre_start() to check if we need to run Application.
  3. Copy any functions marked for RAM to SRAM
  4. Copy data sections to SRAM
  5. Copy .bss sections to SRAM
  6. Call SystemCoreClockUpdate
  7. Call main()

1.7 RAM Section Layout

The linker file, stm32f30_flash.ld contains RAM layout instructions for the linker. Based on guidelines from EmbeddedGurus.com Stack and Heap are placed such that overflows will cause hard fault.

RAM Organization

- **0x20000000 ->** **Main_Stack_Size** Stack pointer is initialized to 0x20000000 + Main_Stack_Size and grows 'down' toward 0x20000000. If stack overflows beyond 0x20000000 Hard Fault Exception occurs: in Debug this will result in bkpt and infinite loop, in Release this will result in NVIC_SystemReset
- **Data Sections**
- **BSS**
- **Heap ->** **0x20010000** Heap grows 'up' toward 0x20010000. If heap overflow occurs Hard Fault Exception occurs: in Debug this will result in bkpt and infinite loop, in Release this will result in NVIC_SystemReset.

1.8 Portability Considerations

There are three main levels of possible dependencies to hardware:

- **Module / Application:** This level only has dependencies on application specific code and isn't coupled to the hardware.
- **Peripheral:** These sections of code are coupled to the vendor's library and are likely similar across a single vendor's products but require changes when switching vendors.
- **Hardware:** These sections of code configure low-level hardware (pin mapping and initialization) and will likely change when moving micros, even within a single vendor.

HAL_PPP_MspInit() and HAL_PPP_MspDeInit() functions are utilized to concetrate HW level configuration to a single file. It is unlikely that other Manufacturers have the same mechanism, but similar files could be utilized.

1.9 Editing this page

This is a barebones Main Page for Terminus documentation. It is stored in **README.md** within the main Terminus folder and is included when Documentation is built. **README.md** can be edited with any text editor and supports Markdown Syntax.