16-bit Communication Protocol

List of commands in sequence as UBHA

* GET\_MEMORY\_ADDRESS\_RANGE = 0x0B
* READ\_VERSION = 0x00
* ERASE\_FLASH = 0x03
* WRITE\_FLASH = 0x02
* READ\_FLASH = 0x01
* RESET\_DEVICE = 0x09
* Optional Command
* CALC\_CHECKSUM = 0x08
* ELF\_VERIFY = 0x0A

Responses sent by the bootloader to a command

* SUCCESS = 0x01
* UNSUPPORTED\_COMMAND = 0xFF
* BAD\_ADDRESS = 0xFE
* BAD\_LENGTH = 0xFD
* VERIFY\_FAIL = 0xFC

1.17.1 Get Version Command

The Get Version command does two things. As the first command that is sent to the bootloader, it is used to establish communication with the device. Therefore, if the communications channel is not setup correctly, this command will fail. Secondly, it returns the device information such as:

- Bootloader version

- Max Packet size

- Device ID

- Erase page size

- Minimum write size

- Config words

**Get Version Command Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x00) - Get Version Command |  |
| 2 | Length | uint16\_t | Unused - Set to 0x0000 |  |
| 4 | Unlock Seqeunce | uint32\_t | Unused - Set to 0x00000000 |  |
| 4 | Address | uint32\_t | Unused - Set to 0x00000000 |  |

**Get Version Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x00) - Get Version |  |
| 2 | Length | uint16\_t | 0x0000 |  |
| 4 | Unlock Sequence | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x00000000 |  |
| 2 | Bootloader Version | uint16\_t | 0x0600 The version of the bootloader |  |
| 2 | Max Packet Size in Bytes | uint16\_t | Maximum size of any packet in either direction. This includes all header and payload data |  |
| 2 | Not Used | uint16\_t | 0x0000 |  |
| 2 | Device ID | uint16\_t | 0x3456(user defined) A generic version number for the user to use |  |
| 2 | Not Used | uint16\_t | 0x0000 |  |
| 2 | Erase Page Size in Bytes | uint16\_t | Size of a erase page on the device in bytes including phantom bytes. This will vary from device to device and the value can be found if the device manual. |  |
| 2 | Minimumn Write Size in Bytes | uint16\_t | The minimum amount of data in bytes that can be written. This also defines the alignment of the data. So if the min write size is 8, then the data must also be 8 byte aligned |  |
| 4 | Not Used | uint32\_t | 0x00000000 |  |
| 4 | User Reserved Area Start Address | uint32\_t | 0x00000000 - Currently not supported |  |
| 4 | User Reserved Area End Address | uint32\_t | 0x00000000 - Currently not supported |  |

**Example Sequence**

Get Version Command - All values in Hex

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x00 |  |
| 2 | Length | uint16\_t | 0x0000 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x000000 |  |

Example command as viewed on bus: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

Example command as viewed on DL: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

**Example Response Sequence**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x00 |  |
| 2 | Length | uint16\_t | 0x0000 |  |
| 4 | Unlock Sequence | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x000000 |  |
| 2 | Version | uint16\_t | 0x1012 |  |
| 2 | Max Packet Size | uint16\_t | 0x0100 |  |
| 2 | Unused | uint16\_t | 0x0000 |  |
| 2 | Device ID | uint16\_t | 0x3456 |  |
| 2 | Unused | uint16\_t | 0x0000 |  |
| 2 | Erase Page Size in Bytes | uint16\_t | 0x0800 |  |
| 2 | Minimumn Write Size in Bytes | uint16\_t | 0x0008 |  |
| 4 | User Reserved Start | uint32\_t | 0x00000000 |  |
| 4 | User Reserved End | uint32\_t | 0x00000000 |  |

Example response as viewed on bus: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x01, 0x00, 0x01, 0x00, 0x00, 0x56, 0x34, 0x00, 0x00, 0x00, 0x08, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

Received response as viewed on DL: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x01, 0x00, 0x01, 0x00, 0x00, 0x56, 0x34, 0x00, 0x00, 0x00, 0x04, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

# 1.17.2 Read Flash Command

This command will read the program flash and return the data read in the response packet. Because of the flash architecture, flash must always be read in modulus 4 byte lengths and the address must also be modulus 4 bytes or on PIC24/dsPIC33 instruction boundaries. The address of the memory range must reside entirely within the application space. If any of the requested data is outside of the application space, a status of 0xFE, Invalid Address, with no data will be returned.

**read\_command\_format Read Command Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x01) - Read Program Memory |  |
| 2 | Length | uint16\_t | Number of bytes to read. Length must be modulus 4 in length. |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to read. Address must modulus 4 bytes. |  |

**Read Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x01) - Read Program Memory |  |
| 2 | Length | uint16\_t | Number of bytes to read. |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to read. Must be modulus of 4 bytes. |  |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |  |

**Command Example Sequence**

Read 8 bytes of data (0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08) starting at address 0x00001878. All values in Hex

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x01 |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00001878 |  |

Example as viewed on bus: 0x02, 0x08, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x78, 0x18, 0x00, 0x00

Provided : 01 04 00 55 00 AA 00 00 1C 00 00

Provided : 01 08 00 55 00 AA 00 00 1C 00 00

**Response Example Response Sequence**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x01 |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00001878 |  |
| 1 | Status | uint32\_t | 0x01 |  |
| 8(for this example) | Read Data from device | uint8\_t | 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08 |  |

Example as viewed on bus: 0x02, 0x08, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x78, 0x18, 0x00, 0x00, 0x01, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08

01 04 00 55 00 AA 00 00 1C 00 00 01 88 1F 04 00

01 04 00 55 00 AA 00 00 1C 00 00 01 88 1F 04 00 00 00 00 00 00

1.17.3 Write Flash Command

This command will program the flash with the data in the payload section. The address and size of the payload will be inspected by the bootloader to prevent accidental over-write of protected spaces. Attempts to write into the memory where the bootloader or configuration bits resides will be prevented and an error will be returned. The flash architecture also places limitations on the address alignment and size of the requests. The start address must always be on an Min Write Size aligned address and its length must also be modulus the Min Write Size in bytes. If either address or length is not aligned the device will not write the data and a status of 0xFE (Invalid Address) will be returned. The user is responsible for erasing flash before writing. Failure to do so will have unexpected results. The correct unlock sequence must be sent to the part for each command as it is not stored on the device.

|  |
| --- |
| **Note** : The size(Length) 0xF0 comes from the fact that the maximum write should not be longer than the max\_packet\_size- command\_size. It should also be a multiple of block\_write\_size. On our case, the max packet size is defined in the bootloader as 0x100(256 in decimal) in bootconfig.h file of the bootloader project. And the write\_flash() command size is 11 bytes. [cmd(1)+len(2)+unlock\_seq(4)+address(4)]. The block write size is 8 in our case. It is defined in boot\_private.h file. So, the max packet size-command size= 256-11=245. The nearest and lower number that is multiple of 8 is 240. Hence the size, 0xF0 is taken by the UBHA.The last page of flash ends before 0xF0 bytes, hence a lower multiple is chosen to accommodate the memory. |

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x02) - Write Program Memory |  |
| 2 | Length | uint16\_t | Number of bytes to program. Must be evenly divisible by the Minimum Write Size parameter in the Get Version response. Command will return error if it's not evenly divisible |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to program. Must be aligned to the Minimum Write Size parameter in the Get Version response. Command will return error if it's not aligned |  |
| Variable | Data To Write | uint8\_t | Seqeunce of bytes. Data to write |  |

**Write Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x02) - Write Program Memory |  |
| 2 | Length | uint16\_t | Number of bytes to write. |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to write. Must be aligned to and modulus of the Minimum Write Size. |  |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |  |

**Write\_Example Command**

Write 8 bytes of data (0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08) starting at address 0x00001878. All values in Hex

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x02 |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Sequence | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00001878 |  |
| 8(for this example) | Data to write | uint8\_t | 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08 |  |

Example as viewed on bus: 0x02, 0x08, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x78, 0x18, 0x00, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08

02 04 00 55 00 AA 00 00 1C 00 00 00 88 1F 04 00

**Example Response Sequence:**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x02 |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00001878 |  |
| 1 | Status | uint32\_t | 0x01 |  |

Example as viewed on bus: 0x02, 0x08, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x78, 0x18, 0x00, 0x00, 0x01

1.17.4 Erase Flash Command

This command erases the number of flash PAGES in the length field. The actual size of the page depends upon the device being programmed. The address must be the beginning address of the first page to be programmed. In addition, attempting to erase memory outsize the application space will result in a Invalid Address (0xFE) status. The correct unlock sequence must be sent to the part for each command as it is not stored on the device

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x03) - Erase Program Memory |  |
| 2 | Length | uint16\_t | Number of pages to erase. Low Byte First |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to erase. Must be page aligned. Low byte first. |  |

**Erase Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x03) - Erase Program Memory |  |
| 2 | Length | uint16\_t | Number of pages to erase. Low Byte First |  |
| 4 | Unlock Seqeunce | uint32\_t | Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Address of the first memory location to erase. Must be page aligned. Low byte first. |  |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |  |

**Example Sequence**

Erase two pages of flash starting at address 0x000A000. All values in Hex

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x03 |  |
| 2 | Length | uint16\_t | 0x0002 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00A000 |  |

Example as viewed on bus: 0x03, 0x02, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x00, 0xA0, 0x00, 0x00

<erase\_command\_example\_response Erase Example Response Sequence

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x01 |  |
| 2 | Length | uint16\_t | 0x0002 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00AA0055 |  |
| 4 | Address | uint32\_t | 0x00A000 |  |
| 1 | Status | uint32\_t | 0x01 |  |

Example as viewed on bus: 0x03, 0x02, 0x00, 0x55, 0x00, 0xAA, 0x00, 0x00, 0xA0, 0x00, 0x00, 0x01

1.17.5 Checksum Command

This command will cause the device to perform a checksum and return the results. The specific checksum will depend on the device and the algorithm used.

**NOTE:** Supporting this command is discouraged and is disabled by default in the MCC Bootloader UI. It is recommended that designs use the SELF\_VERIFY command for integrity checking instead. This command can be used to read out the contents of the device memory even if the Read command is disabled. It is also a weak form of integrity check. The SELF\_VERIFY command can provide more robust integrity and authenticity checks without leaking the device memory.

**NOTE:** The maximum size for the checksum check is 64KB. When validating larger memory ranges, multiple checksum command requests are required.

| **Field Size** | **Description** | **Data Type** | **Comments** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x08) - Calculate and Read Checksum of Program Memory |
| 2 | Length | uint16\_t | Length of the range to be used for the calculation. |
| 4 | Unused | uint32\_t |  |
| 4 | Address | uint32\_t | The start address where to start the checksum. This should be aligned to a PC instruction address in the hex address space. This address is twice the PC address found in the datasheets but match the addresses found in the .hex files. |

**Checksum Command Response Format:**

| **Field Size** | **Description** | **Data Type** | **Comments** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x08) - Calculate and Read Checksum of Program Memory |
| 2 | Length | uint16\_t | Length of the range to be used for the calculation. |
| 4 | Unused | uint32\_t |  |
| 4 | Address | uint32\_t | The start address where to start the checksum. This should be aligned to a PC instruction address in the hex address space. This address is twice the PC address found in the datasheets but match the addresses found in the .hex files. |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |
| 2 | Checksum | uint16\_t | Checksum of the requested memory range. |

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x09 |
| 2 | Length | uint16\_t | 0x0000 |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |
| 4 | Address | uint32\_t | 0x00000000 |

**Checksum Example Response Command**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |
| --- | --- | --- | --- |
| 1 | Checksum command | uint8\_t | 0x09 |
| 2 | Length | uint16\_t | 0x0000 |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |
| 4 | Address | uint32\_t | 0x00000000 |
| 1 | Status | uint8\_t | 0x01 (Success) |
| 2 | Checksum Result | uint16\_t | 0x55AA |

Example as viewed on bus: 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0xAA, 0x55

1.17.6 Reset Device Command

This command will cause the device to do a software device reset. The reset will occur right after the last byte of data is set out of the UART

| **Field Size** | **Description** | **Data Type** | **Comments** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x09) - Reset Device |
| 2 | Length | uint16\_t | 0x0000 - Field Ignored |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 - Field Ignored |
| 4 | Address | uint32\_t | 0x00000000 - Field Ignored |

**Reset Command Response Format:**

| **Field Size** | **Description** | **Data Type** | **Comments** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x09) - Reset Device |
| 2 | Length | uint16\_t | 0x0000 - Field Ignored |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 - Field Ignored |
| 4 | Address | uint32\_t | 0x00000000 - Field Ignored |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |

**Reset Example Command:**

Example Reset Device command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x09 |
| 2 | Length | uint16\_t | 0x0000 |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |
| 4 | Address | uint32\_t | 0x00000000 |

Example as viewed on bus: 0x09, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

**Example Response Sequence**

Example response from the device

| **Field Size** | **Description** | **Data Type** | **Value** |
| --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x09 |
| 2 | Length | uint16\_t | 0x0000 |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |
| 4 | Address | uint32\_t | 0x00000000 |
| 1 | Status | uint32\_t | 0x01 |

Example as viewed on bus: 0x09, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01

1.17.7 Self Verify Command

**Self Verify Program Memory Command**

This command will cause the device to verify the contents of its flash by computing the Checksum/CRC32/SHA256 of the application program and compare the computed value to the expected value located in the application program header. If the compared contents match, a Success value is returned. If they do not match, an Invalid Compare value is returned.

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x0A) - Self Verify Program Memory |  |
| 2 | Length | uint16\_t | 0 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0 |  |
| 4 | Address | uint32\_t | 0 |  |

**Self Verify Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x0A) - Self Verify Program Memory |  |
| 2 | Length | uint16\_t | 0. |  |
| 4 | Unlock Seqeunce | uint32\_t | 0 |  |
| 4 | Address | uint32\_t | 0 |  |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address * 0xFD Invalid Compare |  |

**Self Verify Example Command**

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x0A |  |
| 2 | Length | uint16\_t | 0x0000 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x00000000 |  |

Example as viewed on bus: 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

**Example Response Sequence**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x0A |  |
| 2 | Length | uint16\_t | 0x0000 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x00000000 |  |
| 1 | Status | uint32\_t | 0x01 |  |

Example as viewed on bus: 0x0A, 0x00, 0x00, 0x00 0x00, 0x00 0x00, 0x00 0x00 0x00, 0x00, 0x01

1.17.8 Get Memory Range Command

This command will request the memory range for a specific memory region on the device. The specific memory region will be placed in the "Address" field. This field should be populated with 0 if only one memory region is supported. The device will respond with the start and end address of the programmable memory range. This will allow the programmer to determine what ranges are supported and allow the programmer to filter out all other memory addresses.

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uintS8\_t | Command (0x0B) - Get Memory Address Range |  |
| 2 | Length | uint16\_t | 0x8. Length of the response. |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00 |  |
| 4 | Address | uint32\_t | Which memory range. 0x0 is default for a divice with just single flash. |  |

**Get Memory Address Range Command Response Format**

| **Field Size** | **Description** | **Data Type** | **Comments** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | Command (0x0B) - Get Memory Address Range |  |
| 2 | Length | uint16\_t | 8 -Number of bytes to read. |  |
| 4 | Unlock Seqeunce | uint32\_t | 0 Unlock sequence for flash. Key for currently supported parts is 0x00AA0055 |  |
| 4 | Address | uint32\_t | Which memory range. 0x0 is default for a divice with just single flash. |  |
| 1 | Status | uint8\_t | Status of Command   * 0x01 Success * 0xFF Unsupported command * 0xFE Invalid Address |  |

**Get Memory Address Range Example Sequence**

Get Address Range of flash. StartAddress = 0x2000 End Address = 0AB7FE. All values in Hex

Example command to the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x0B |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x00000000 |  |

Example as viewed on bus: 0x0B, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

**Get Memory Address Range Example Response**

Example reponse from the device

| **Field Size** | **Description** | **Data Type** | **Value** |  |
| --- | --- | --- | --- | --- |
| 1 | Cmd | uint8\_t | 0x0B |  |
| 2 | Length | uint16\_t | 0x0008 |  |
| 4 | Unlock Seqeunce | uint32\_t | 0x00000000 |  |
| 4 | Address | uint32\_t | 0x00000000 |  |
| 1 | Status | uint08\_t | 0x01 |  |
| 8(for this example) | Read Data from device | uint8\_t | 0x00,0x20,0x00,0x00,0x00,0x0A, 0xB7, 0xFE |  |

Example as viewed on bus: 0x0B, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x20, 0x00, 0x00, 0xFE, 0xB7, 0x0A, 0x00

# 1.18 Flash Erase/Write Operation Security

**Issue Overview**

Whenever code has the ability to change the flash content, those operations need to be safe guarded as much as possible against accidental writes/erases. Once source of accidental operation execution is caused by invalid code jumps. These could be caused by software bugs or hardware issues like miss-execution from improper BOR guarding at system power up, power down, or a power spike/glitch.

Below is an example code code for writing some data to flash that we'll use for discussion:

**Example 1**

#define FLASH\_UNLOCK\_KEY 0x00AA0055u

**if**(UpdateNeeded() == true){

**if**(dataValid() == true){

FLASH\_Unlock(FLASH\_UNLOCK\_KEY);

FLASH\_Write(data);

FLASH\_Lock();

}

}

In example 1 above the flash unlock key is hard coded into the program. The validity checks of if we want to write the data into flash are done before unlocking the flash. In normal operational conditions, this code looks like it would be very reasonable. But in the miss-execution situations that we mentioned previously, this code is actually dangerous. Consider the case when a miss-executing code jumps from somewhere else in program memory to line 3. If the code accidental jumps to that line a write to the flash is going to happen, possibly with invalid data.

As you can see this code has some risk around the flash operations.

**Mitigation Techniques**

There are a few methods that can be used to help mitigate some of the risk around these flash operations. We will cover the techniques used this bootloader solution to help reduce that risk.

**Externally Sourced Key:**

The first mitigation technique is not hard code the key, but instead get the key from an external source that is not always present. For example, for the flash operations associated with downloading a new flash image by an external bootloader host, that host can provide the key in the command data to the target device. We use this technique for all of our host driven flash operations. In this case the flash unlock key isn't in the code and thus it becomes very unlikely that a flash operation can occur when you aren't in a firmware upgrading session. Below is an example of what this might look like.

**Example 2**

uint8\_t commandPacket[100] = {0};

uint32\_t flashKey = 0;

UART\_GetCommand(&commandPacket); //Read command from host

memcpy(&flashKey, commandPacket[4], **sizeof**(uint32\_t));

FLASH\_Unlock(FLASH\_UNLOCK\_KEY);

FLASH\_Write(data);

FLASH\_Lock();

memset(commandPacket, 0x00, **sizeof**(commandPacket));

In example 2, the host is sending the flash key as part of the protocol. As a result, the target device code doesn't have a copy of the key at all. So if we are outside of a bootload operation and a command hasn't been sent by the host, then a flash operation is much less likely.

Let's say we are outside for the bootload operation and because of a code error, the code jumps to line 6 of example 2. In this case the flash will only unlock if the commandPacket[4]-commandPacket[7] have the unlock key still in it. After each flash operation, we destroy that copy of the key from RAM (line 9 in example 2). So unless the errant code managed to not only jump to this location, but also write the correct 32-bit value to the right location, then the flash write operation will be blocked. This greatly reduces the risk of an accidental flash write.

**Calculated Flash Key:**

There are times when you may need to do flash operations when not connected to an external source that can limit the devices access to the flash key to only when it is allowed. In this case another method to safe-guard the key must be used.

This bootloader solution uses a calculated flash key to help reduce the risk of invalid key usage.

We saw in example 1 that the miss-execution case allowed the code to by-pass all of the validity checks for the write operation that we had in place in example 1 lines 1 and 2. To prevent that from happening, instead of supplying the entire key in one place, we provide part of the key in two places - a part of it before the validity checks and the second part after the validity checks. This makes it so that the code must go through the validity checks in order to do the flash operation. Let's look at an example of what this might look like.

**Example 3**

#define FLASH\_UNLOCK\_KEY 0x00AA0055u

#define PARTIAL\_KEY\_1 (FLASH\_UNLOCK\_KEY + PARTIAL\_KEY\_2)

#define PARTIAL\_KEY\_2 (0x12345678u)

**volatile** uint32\_t flashKey = 0;

flashKey = PARTIAL\_KEY\_1;

**if**(UpdateNeeded() == true){

**if**(dataValid() == true){

FLASH\_Unlock(flashKey - PARTIAL\_KEY\_2);

FLASH\_Write(data);

FLASH\_Lock();

}

}

flashKey = 0;

In the example above we use the real flash key to create two partial keys that are used together to calculate the real key. We use these partial keys rather than the real key. We place one key before the validity checks (line 4) and we place the other after the validity checks (line 7).

Just like with example 1, let's say the miss-executing code jumps to the unlock call at line 8. By default the flashKey variable is 0 and we always reset it to 0 after a flash operation (line 13). So if the flashKey variable is 0, then the FLASH\_Unlock() function gets called with the value 0x12345678 and thus will not unlock the flash since a value of 0x00AA0055 is required to unlock the flash.

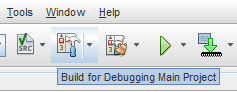
What happens if we jump to line 5 instead when we don't want to update the flash? The flashKey will be loaded with the first half of the key. But the code will now be forced to run through the validity checks. If these validity checks are able to determine that a flash write is not desired, then an invalid flash write can be prevented and the flashKey is set back to 0 to help prevent other bad writes.

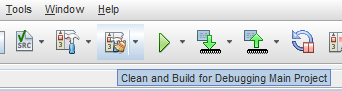
This doesn't completely prevent the invalid flash operations occurring, but it helps reduce the likelihood of it happening.

**NOTE:** In the example above we declared the flashKey variable volatile to help reduce the likelihood that the compiler would optimize the flashKey calculation all into line 8. If the compiler determined that the flashKey was a constant value and it could just inject that constant value in the FLASH\_Unlock() function directly instead of creating a variable, then it would break the safeguard we just tried to put in place.

1.19 Building for Production

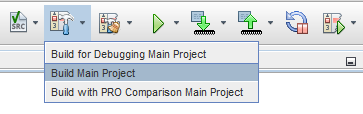
In some version of MPLAB X, the 'Build' and 'Clean and Build' buttons located in the top toolbar will run in debug mode by default. When built in debug mode, a .hex file will **not** be generated under the project's dist\default\production directory which is required for verification signatures and merging with the bootloader.

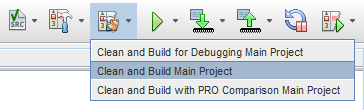




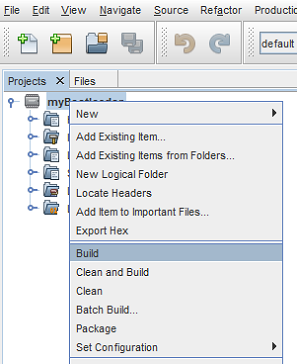
To produce a .hex file and build in production mode, please choose one of the following options:

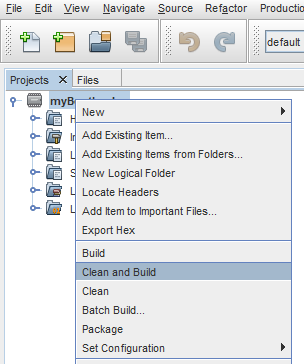
* Select the dropdown next to the 'Build' or 'Clean and Build' button and select 'Build Main Project' or 'Clean and Build Main Project' respectively

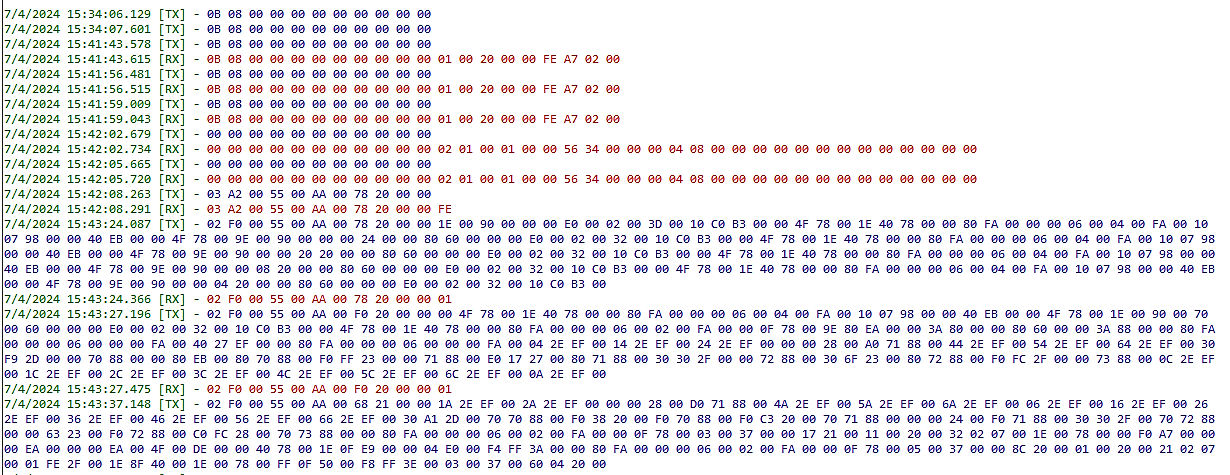
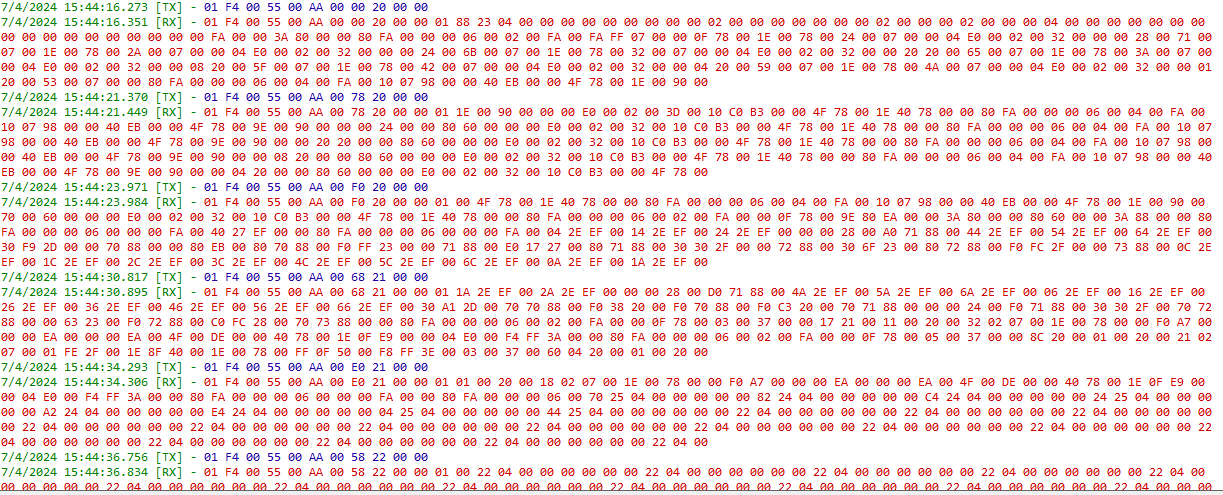
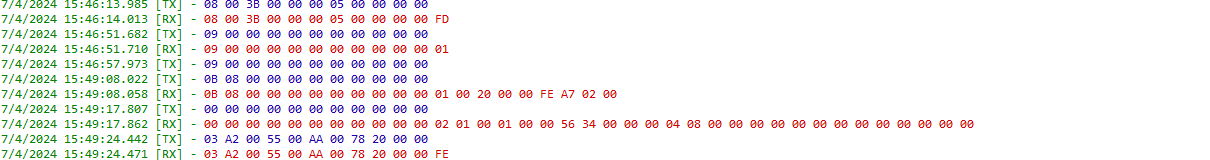




* Right click on the project name in the Projects window and select 'Build' or 'Clean and Build'







Reference :

<https://onlinedocs.microchip.com/oxy/GUID-4F4EDA17-350C-4EC3-B601-9F459DD028EF-en-US-5/index.html>