BOOTLOADER

# INTRO

1. In order to support bootloader (aka BL) features, an MCU program memory shall be designed with a specific initial (or sometimes final) section dedicated for the bootloader code.
2. For dsPIC33 MCUs the program memory (aka flash) is typically divided into x5 sections: RESET, IVT, BOOTLOADER, APPLICATION and CONFIGURATION.
3. For typical applications, the bootloader and IVT (aka Interrupt Vector Table) code are programmed only at production time and remain fixed (often remains the same even among different products), whereas the actual application code (including where ISRs implementation is written) can be changed in time (e.g. to update the code or fix a bug). To allow this, a possible solution is to create a "user remap IVT" in the application memory (and keeping the address-location of this RAIVT and of all of its entries fixed). However, there might arise issues if both bootloader and application software try to use same interrupts! Microchip's solution is the "Code-Guard" feature: when enabled, it creates a 2nd "virtual" IVT in the application memory (named "Alternative IVT", independent of the primary IVT and thus not part of the bootloader section), so that while booting the bootloader code uses the primary IVT, and then (after switching to nominal operation) the application software uses the AIVT.
4. The x4 main tasks of the bootloader kernel are:
   1. take care of the boot procedure;
   2. check for application updates (through buttons, UART, etc);
   3. verify the integrity of the application image before starting it (via CRC, SHA256, etc);
   4. transfer program flow from bootloader level to application image level after boot procedure is completed.
5. The most important bootloader-memory section is further divided into several sub-sections, where the most important one is the [A] bootloader-kernel (in charge for all the previous tasks). The kernel shall implement a set of mandatory commands:
   1. ***erase flash*** to erase a desired number of pages in the application memory (i.e. only if the specified address range reside in the application memory);
   2. ***program flash*** to write pages of the application memory;
   3. ***read version*** to check the bootloader version;
   4. ***read memory settings*** to check start/final address of the application memory);
   5. ***self-verify*** to check the application image consistency by means of the specified method (CRC, SHA256, etc).
6. Other additional commands (which can be either enabled or disabled) are:
   1. ***read flash*** to read specified range of program memory (which can also cause security risks in terms of intellectual property stealing);
   2. ***reset device*** to force a soft-reset of the device.
7. Besides the kernel, the bootloader section has other sub-sections (in the program memory): *verification driver* to check and approve/reject code verification using the specified method, and *communication driver* to communicate with the host through specific interface (UART, USB, I2C, etc).
8. The application memory section (in the program memory) is divided as well into several sub-sections: *application certificate* involved in the verification process, *user remap IVT* (or AIVT for Microchip), and *application code* independent of bootloader usage.

# SECURE BOOTLOADER

1. Nowadays the x2 main security features (also covered by MPLAB) are:
   1. ***Integrity*** to be sure the firmware image has not been altered (due to either flash failure or intentional hacker attack) by calculating a checking binary string (called "digest") through some cryptographic hash functions (e.g. SHA-256)
   2. ***Authenticity*** to be sure the code comes from a valid source (since someone could upload their own code with their correct digest value and the integrity check alone would not detect the issue) by a signature mechanism of the digest using some signing algorithm (e.g. ECDSA) fed with a private key (so that only someone knowing the private key can generate that specific signature). Then the signature is attached to the application image so that anyone can check the firmware authenticity through the public key (thus a pass/fail authentication result can be executing giving the digest, the public key and the image signature as input to the ECDSA function).
2. In particular, in MPLAB specific implementation, the application image (besides the actual application code) contains an initial header section to store the signature and some additional metadata (e.g. code start/end address). To generate the signature, first it is calculated the digest of metadata plus application code sections (e.g. with SHA256), then this digest is given as input to the ECDSA function along with the private key to produce the signature that is finally added in its specific location in the aforementioned application header.

# DEMO-PROJECT

## Notes

* **BTN** to force bootloader into the application update mode (actually not needed, just connect this to GND to enter in bootloader mode while booting and enable communication with UBHA, otherwise keep it connected to VCC to execute bootloader initialization and then switch to application): remapped from RA7 to RC15 (PWMRC);
* **Yellow LED** to signal bootloader is running: remapped from RC3 to RB15 (EXTR1);
* **Blue LED** as application code #1 operation: remapped from RC5 to RB14 (EXTR3);
* **Green LED** as application code #2 operation: mapped to RB12 (EXTR5);
* **UART** to download new firmware in bootloader mode: remapped from RC8/RC9 to RB6/RB5 (U1TX/U1RX).
* Available bootloader verification methods from MPLAB and plug-ins : Flash-not-blank, CRC32 or SHA256.
* Use 220 Ω resistor as LED series resistance.
* For MCU models not supported by MCC, Microchip provide another lighter (and less robust) bootloader tool called EZBL.
* All steps assume to use a dsPIC33CH512MP508 MCU (e.g. PXS board), MPLAB X IDE v5.45 and XC16 v1.61.

## Bootloader project

Create the bootloader project:

1. Create new standalone project (e.g. named ExBootloader.X);
2. Open MCC, go to *Pin Manager > Pin Module* and set RC3 as OUTPUT, RA7 as INPUT;
3. Go to *Device resources > Libraries > 16-bit Bootloader* and double-click *16-bit Bootloader : Bootloader*. On the *Bootloader Settings* tab opened choose the communication protocol (e.g. UART1 FSL), tick desired commands to be included (e.g. RESET, READ, etc), choose the verification scheme (e.g. CRC32);
4. Go back to *Pin Manager > UART1* and assign U1TX to RC8 and U1RX to RC9 (plus U1TX as idle-high and U1RX with pull-up, if needed);
5. Go to *MCC > Project Resources* and press *Generate* (to check warnings go to *Notification* tab);
6. Close MCC;
7. Add code for button handling in function *EnterBootloaderMode*,
8. Add code for blinking yellow LED while in bootloader mode in function *BOOT\_DEMO\_Task*;
9. Compile and flash code;
10. Check yellow LED is actually blinking in the board.

## Application project

Create application project #1:

1. Create new standalone project (e.g. named ExApplication1.X);
2. Open MCC, go to *Pin Manager > Pin Module* and set RC5 as OUTPUT;
3. Go to *MCC > Project Resources* and press *Generate* (to check warnings go to *Notification* tab);
4. Close MCC;
5. Add code for blinking blue LED while application is running in function *main*;
6. Compile and flash code;
7. Check blue LED is actually blinking in the board (and yellow LED stops blinking).

## Application-to-Bootloader linking

Now link the application project #1 to the bootloader via MCC:

1. Load *ExApplication1* project in MPLAB;
2. Open MCC;
3. Go to *Device resources > Libraries > 16-bit Bootloader* and double-click *16-bit Bootloader : Application.* On the *Easy Setup* tab opened specify the bootloader project path (e.g. *../ExBootloader.X*)
4. Go to *MCC > Project Resources* and press *Generate* (to check warnings go to *Notification* tab). Note this create a new folder *boot* with several files inside *MCC Generated Files*;
5. Close MCC;
6. Trying compiling should generate a warning/error saying “A script to append verification checksum…”. Comment out the #warning/error line and copy the line below (e.g. *cd mcc\_generated\_files/boot && postBuild.bat $(MP\_CC\_DIR) ${ProjectDir} ${ImageDir} ${ImageName} ${IsDebug}*) in to *Project > Properties > Conf > Building > … after build* field and tick *Execute this line* after build;
7. Compile again and flash the MCU;
8. Check blue LED is still blinking (but now also the Bootloader settings have been flashed and they are ready to be used).

## Read/write from bootloader mode through UBHA

Now use the *Unified Bootloader Host Application* (aka UBHA) to read/write application code through the bootloader:

1. Set *ExBootloader* project as main in MPLAB and flash the MCU (blue LED should stop blinking and yellow one start). So now the MCU is “virgin”, thus just the bootloader is running with no application;
2. Open the *Unified Bootloader Host Application*;
3. Set *Device Architecture* as *PIC24/dsPIC33*;
4. Go to *Setting > Serial*, set *COM* connected to USB-to-TTL adapter (e.g. COM3) and baud rate (e.g. 9600 b/s);
5. Press the *Read Device Settings* button to automatically retrieve the MCU application code start/end addresses (NB: this is essential!);
6. Regarding the un/tick-options, when enabled (and if supported/enabled as well by the MCU bootloader), the 1st forces the MCU to reset after being programmed, the 2nd to force a read-back verification from MCU after being programmed (and check if matches with *.hex* file on PC), and the 3rd forces the bootloader to perform a self-verification (just like while booting) and send the pass/fail result back to the UBHA. All three can be enabled for this example;
7. Go to *File > Open/Load File* and choose the desired application image to flash through the bootloader (e.g. in *ExApplication1.X/dist/default/production/ ExApplication.X.production.hex*);
8. Press *Program Device* to flash the MCU via UART and bootloader, and check the blue LED starts blinking;
9. For verbose info about commands and responses, see *Tools > Console*.
10. Now in order to reprogram again the MCU this shall be put in bootloader mode again. To do so, connect the BTN pin to GND and perform an MCU reset (e.g. just a soft one by pulling up/down the MCLR line). Now the yellow LED should start blinking signalling the MCU is in bootloader mode. Now reconnect the BTN pin to VCC and try transmitting ExApplication2 via UBHA. The green LED should then start blinking.

## Further projects

* To create additional application projects (e.g. #2) able to work with the existing bootloader, just repeat the steps descripted in chapters 4.2 and 4.3;
* To create new bootloaders (for example, to use a different authentication method), repeat the steps described in chapter 4.1.

## Secure-bootloader upgrade

So far with CRC32 or SHA256, only verification checks have been performed on the application images. To perform also signature authentication the ECDSA algorithm is needed. However, due to its complexity an external chip is needed to perform these calculations: ATECC608A by Microchip, communicating with the MCU via I2C.

# REFERENCES

* [Tutorials](https://mu.microchip.com/16-bit-bootloaders-using-mcc-device-side)
* [Documentation](https://www.microchip.com/en-us/software-library/dspic33-pic24-bootloader)
* [Further info](https://ww1.microchip.com/downloads/aemDocuments/documents/MCU16/ProductDocuments/ReleaseNotes/release_notes_bootloader_1_22_1.html#WhatIsThis)
* [EZBL](https://www.microchip.com/SWLibraryWeb/product.aspx?product=Microchip%20Easy%20Bootloader)
* [GitRepo](https://github.com/microchip-pic-avr-examples/bootloader16-microchip-university-labs)

# TODO

* Be sure DUAL\_PARTITION option is enabled (?)
* No interrupt needed by bootloader?
* Try uploading a new application code image (for example, to blink new green LED) just with bootloader.