

GSoC 2020 Final Report

apertus° Association

AXIOM Remote: Bootloader improvement and
extension

Priya Pandya

Mentor: Andrej Balyschew

Co-mentor: Herbert Poetzl

August 31, 2020



Contents

1	Introduction	1
2	Work Product	1
3	Tasks Implemented	1
3.1	Phase I	1
3.1.1	Host PC script	1
3.1.2	ProcessCommand()	2
3.2	Phase II	2
3.3	Phase III	2
3.3.1	CRC8 class and Sequence number	2
3.3.2	GetInfo()	3
3.3.3	SelectChip()	3
3.3.4	GetKMData() and WriteData()	3
4	Challenges Faced	4
4.0.1	Phase I	4
4.0.2	Phase II	4
4.0.3	Phaase III	5
5	Conclusion	5

1 Introduction

As the title says, my work is to improve the functionality of the existing Bootloader and add more features to it. The task is described [here](#) The main feature to be implemented was **Flashing of PIC16 firmware using PIC32 Bootloader** and some other features:

- PIC32 self-programming
- Communication between Bootloader and Firmware
- Power saving modes for PIC32

2 Work Product

All the code produced by me during GSoC 2020 can be found in this [repository branch](#).

3 Tasks Implemented

3.1 Phase I

The following tasks were implemented in the first phase, along with detailed understanding and reading up about the project:

3.1.1 Host PC script

The main script runs on the host PC, which is in python, has all the functionalities to select the chip, the mode (read or write), the HEX file to be flashed etc.

The command-line options/arguments are:

- **port** : path to the USB port
- **hex** : path to the HEX file to flash
- **chip** : to select the microcontroller chip, it has three options: km_west, km_east, pic32
- **version** : display the versions of the firmware
- **read** : to read code back from the PIC memory
- **write** : write the HEX code in the PIC memory

The HEX file is processed using the **intelhex** python library The reading and writing to and from the USB port is done using **pyserial** To communicate with the Bootloader, an ASCII communication protocol is used which has some commands which are sent over to the Bootloader and it sends back acknowledgement (negative and positive both).

3.1.2 ProcessCommand()

The class USBDCDDevice already had the methods to read and send the data from the Bootloader to the PC via the USB port connection. I've added a function ProcessCommand() in the main.cpp of the firmware which receives the command string from the PC and checks it and accordingly decides whether to send ACK or NACK. An additional callback function is also implemented there and a setter for that.

3.2 Phase II

The progress for this phase was as follows:

- The connection between the host PC and the remote is successfully established.
- Some changes are made to the python script developed in the Phase-I, now the Set/Get command, every sequence number digits, checksum part, $\langle RS \rangle$, $\langle EOT \rangle$ and every character of fields take 1 byte each. The reason I did this is because if we consider $\langle EOT \rangle$ value as "0x4" then it's length will be 3 which takes up too much space and iteration will increase in the ProcessCommand() so instead of that I'll take it's ASCII character so it'll only take 1 byte space.

To explain this better, here is an example to know the difference between the previous command string and the current one:

Previous: "S0014d0x1epic320x4", it takes 18 bytes space

Current: "S1M▲pic32◆", it takes 10 bytes space

- The ProcessCommand() method on the remote side is now broken-up into many sub-routines so as to ease the process and de-clutter the code.

3.3 Phase III

In the last phase much progress was done:

3.3.1 CRC8 class and Sequence number

I added the CRC8 class on the remote side to check the CRC8 of the data coming from the host PC. To send a positive acknowledgement to the host PC, both the sequence number and the checksum need to be the same as the packet. This validation was successfully done in this phase.

3.3.2 GetInfo()

This method is added in the main.cpp. Its function is to extract the useful information from the packet received from the host PC then check the sequence number and CRC8 and send an acknowledgement. This also calls another method **SelectChip()**.

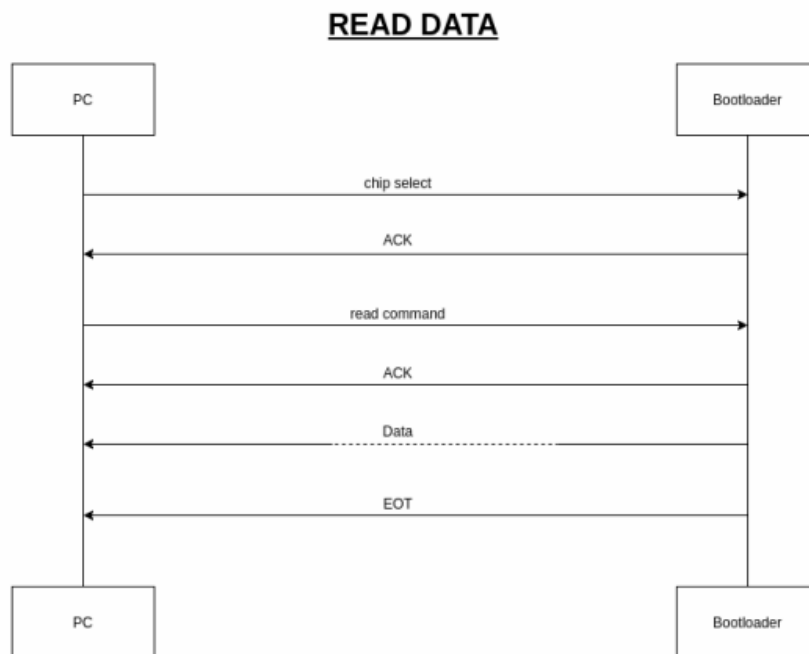
3.3.3 SelectChip()

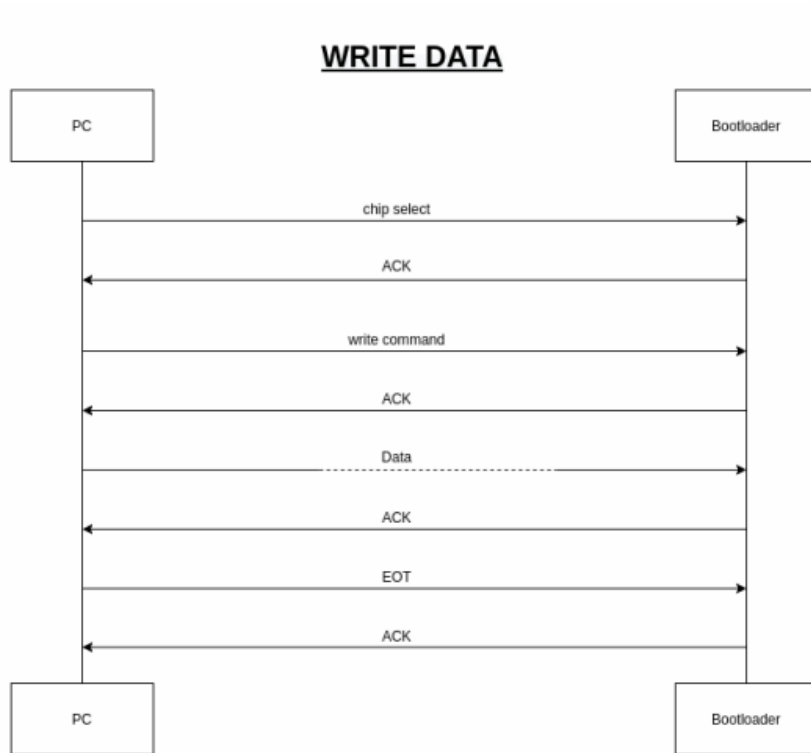
This takes the chip name as the argument and then uses the ICSPProgrammer class to activate that respective chip. It calls the **GetKMData()** or **WriteData()** method depending upon the operation (read or write).

3.3.4 GetKMData() and WriteData()

These use the ICSPProgrammer class's functions (EnterLVP(), SendCommand(), ExitLVP() etc.) to read or write the data from or to the selected chip.

The following images demonstrate a simple read and write process between the host PC and the remote:





4 Challenges Faced

4.0.1 Phase I

The main problem which I initially faced was to properly understand the flashing process in which Andrej helped a lot by using various examples. I also realised that while writing the proposal I had underestimated this task. Although it may seem simple that all I need to do is to put a HEX file into the microcontroller's memory but the real challenge is to make the process as user-friendly as possible.

One of the other obstacles I encountered when I wanted to do some tests for the section 2.2 code. The Bootloader HEX file had some problems and was flashing but not creating the USB port so communicating it with the host PC was not possible. So Andrej told to do the testing on the Firmware HEX for now as that worked perfectly and I was also able to do the tests.

4.0.2 Phase II

While trying to send the command string from the host PC to the remote, I discovered that the string of length greater than 8 is not received at the remote's end and the script usually

freezes. After debugging with Bertl and BAndiT we concluded that there is some bug either in the USB code or hardware issue. So the packet is cropped and then sent to the remote.

4.0.3 Phaase III

- CRC8 on the host side (the python script) was giving some trouble when converting to 1 byte length due to which the reading of data was not becoming possible, so I had to do changes to that data byte.
- The intelhex gives data in decimal (int data type) so when sending that data, the length of each data was coming out to be between 1 to 5 bytes so had to make many changes to lower the length of that data which took a lot of time, so now the data is ready on the host side and receiving part is also ready on the remote side.

5 Conclusion

The flashing task is taking time as it may sound simple but the implementation part is quite complex with numerous big and small obstacles. Especially, the integration of the hardware and the software was a little time taking in the beginning. Some of the tasks mentioned in my proposals are still left which I'll be working on after the GSoC period.

My overall experience with apertus°Association was great. I learnt everything from coding styles, microcontrollers, various python manipulations and modules to those low-level parts of C++ which I had never explored. My mentor (Andrej) was very supportive and was always available whenever I was stuck anywhere or could not understand a portion.