# Software Development Approach

The development of the software goes into three stages:

1. Writing the program and emulating it with “6800 Development Environment” which is an emulator the MC6800 processor – these files are with prefix: “emul\_”. This is used to emulate and after that validate the program’s expected behavior.
2. Rewriting the program according to the syntax rules and grammar of “Motorola Freeware PC-Compatible 8-Bit Cross Assemblers” which is an assembler for the family 68xxx – these files are with prefix: “asm\_”. This is used to then generate the hexadecimal codes that represent the instructions of the processor or else said to generate the .s19 file.
3. Converting the .asm files through the program s1rec\_to\_bin.cpp and “Motorola Freeware PC-Compatible 8-Bit Cross Assemblers” to a binary file (Later explained in detail in “Instructions for using s1rec\_to\_bin.cpp”). The binary file is imported into “XP8710 Software” and then loaded into the EPROM of the controller via programmer. Finally, the EPROM is placed into the controller and the developer validates the program`s behavior.

# Validated programs

NOTE: When a program is both emulated and ran on the controller successfully only the one tested on the controller (with prefix: “asm\_”) will be mentioned.

## Programs validated on the controller:

1. asm\_wr\_outs.asm

The program sets the Modem Control Register (MCR) of the UART chip to a value that

sets the pins OUT1 and OUT2 at a state as follows: low and high.

1. asm\_writing\_to\_comp.asm

Firstly, the program initializes the UART to operate with 8-bit data, no parity, and one stop bit, BR of 9600, enables interrupts Receive Data Available RDA and Transmitter Holding Register Empty THRE via the Interrupt Enable Register (IER) (in label \_init\_uart). then writes into the Transmitter Holding Register (THR write only) the sequence 0 to 9 infinitely to the device connected to the controller via the Serial Output pin (SOUT).

The test was implemented with two additional technologies:

* TTL to USB connector
* “PuTTy (64-bit)”

both used for serial communication.

The expected result after executing the program is to visualize infinite sending of the sequence 0 to 9 on the terminal window of PuTTy.

1. asm\_sram\_test.asm

The program loads value into accumulator A (ACCA) of the processor and stores it into an address in the SRAM of the controller chosen by the developer. Then loads the value from the same address in accumulator B (ACCB) and stores the value into the Modem Control Register (MCR) of the UART chip that sets the pins OUT1 and OUT2 at a state as follows: low and high.

The program validates that the SRAM is in good condition and working as expected.

1. asm\_simple\_loop\_back\_comm.asm

Firstly, the program initializes the UART to operate with 8-bit data, no parity, and one stop bit, BR of 9600, enables interrupts RDA and THRE via the Interrupt Enable Register (IER) (in label \_init\_uart) also setting the Stack pointer (SP) to the biggest address available in the SRAM ($1FFF). Then the processor is polling for Data Ready (DR) in Line Status Register (LSR) (implemented with subroutine \_poll\_dr) from the connected to the controller device via USB to TTL connector and through PuTTY. When input arrives DR rises, the program stops the polling subroutine and loads the data from the Receiver Buffer Register (RBR) (read only) in ACCB and pushes it into the Stack then clears ACCB. Forwards the processor is polling for THRE in LSR (implemented with subroutine \_poll\_thre). When THRE rises data is pulled from the Stack and loaded into ACCB and then written to THR.

The program`s result is simple: The user should be able to write into the terminal window of PuTTy and receive the same characters as the ones he sends to the controller.

## Programs validated in the “6800 Development Environment”:

1. emul\_loop\_back\_comm.asm

The program simulates a complete loop-back communication mechanism, where the primary objective is to receive a packet of data from a simulated UART, buffer it in the SRAM, and subsequently transmit the same data back, thereby validating the communication flow before deploying on the controller.

Firstly, the program initializes the UART to operate with 8-bit data, no parity, and one stop bit, BR of 9600, enables interrupts RDA and THRE via the Interrupt Enable Register (IER) (in label \_init\_uart) also setting the Stack pointer (SP) to the biggest address available in the SRAM ($1FFF).

After buffering the data, the program simulates the transmission of the same data back to the simulated UART by polling for the THRE bit in the LSR. Once the transmitter is ready, the data is pulled from the SRAM and sent back.

The program uses a buffer in SRAM to store received data. The buffer has a maximum size of 15 bytes ($0F), and the program keeps track of the current buffer position and capacity.

When the buffer is full or a specific termination character ('9') is received, the program marks the reception as complete and prepares to transmit the buffered data.

The program includes basic error handling for UART communication errors by checking the Interrupt Identification Register (IIR). If an error is detected, the program clears the error and continues execution.

The program simulates an increasing input of the sequence from 0 to 9 to test the loop-back communication mechanism. This simulated input is used to validate the program's ability to handle and transmit data correctly.

The expected result of this program is to successfully simulate the loop-back communication mechanism in the emulator. Any data sent to the simulated UART should be received, buffered, and then transmitted back without errors. This ensures that the communication logic is sound and ready for deployment on the controller.

The program is validated in the "6800 Development Environment" by observing the simulated communication flow. The developer can verify that the data is correctly received, buffered, and transmitted back, ensuring that the program behaves as expected before moving to the controller.

# Instructions for using s1rec\_to\_bin.cpp:

## Prerequisites:

1. Compiler for the programming languages C/C++
2. “Motorola Freeware PC-Compatible 8-Bit Cross Assemblers User's Manual”

It is an open repository at: <https://github.com/JimInCA/motorola-6800-assembler>

If you have the required prerequisites the next steps are:

1. Compile the program with the following command:

g++ -o s1rec\_to\_bin s1rec\_to\_bin.cpp

2. Run the program with the following command:

./s1rec\_to\_bin your\_asm\_file.asm

This will create a file your\_asm\_file.bin in the same directory as the your\_asm\_file.asm file.

The file your\_asm\_file.bin is the translation from the Assembly instructions in your\_asm\_file.asm to their binary representations.

IMPORTANT: If your\_asm\_file.bin is imported directly into the software you use for writing into memory will place its contents at address zero in the memory.