# Reimagining Orchestral Chimes Electronically

Submitted to the School of Computer Science and Technology at Algoma University as a Fourth Year Project submission

Johnny Console
*School of Computer Science and Technology*
*Algoma University*
Sault Ste. Marie, Ontatio, Canada
johnny.console@algomau.ca

***Abstract*—This project aims to produce an electronic clone of an orchestral chime that can be tuned to any sound sample using the included utility software package.**

## I. Motivation

Orchestral chimes are expensive, often costing thousands of dollars for a professional set. This project aims to produce a much cheaper electronic version of these chimes.

## II. Project Goals

The goal of this project is to produce a cheaper version of an orchestral chime set where the chimes are made of hollow tubes. These tubes would then be able to be stricken, causing a pressure sensor in the tube to detect how hard or soft the tube was stricken. The chimes themselves would be controlled by a microcontroller which interfaces with a non-volatile flash memory chip and a digital to analog converter. There would be a pedal whereby the user could press it and all sounds would stop. Due to timing constraints, the flash memory was not included in the final product.

## III. Protocols Used

This project implements two main protocols for its use: one that is designed specifically for this project, and another that is widely used.

### A. *Serial Communication Protocol*

Due to the nature of the enhanced universal serial a receiver transmitter port on the microcontroller chip, a serial protocol to accept changes to the loaded sound sample was created. This serial protocol involves only four instructions: send the next block of data, resend the same block of data (usually due to an error), send the checksum of the most recently sent block of data, and terminate the communication.

A communications protocol between the host computer and the device was designed which functioned in the following way:

1. The server sends the first block of data over the serial bus.
2. The microcontroller sends the checksum command, namely a ‘C’ character to the serial bus.
3. The server calculates and sends the block’s eight-bit checksum value through the serial bus.
4. The microcontroller calculates the block’s eight-bit checksum and compares it to the received value. If equal, the microcontroller sends the block to the flash memory and then sends the command to receive the next block, namely an ‘N’. Otherwise, it will send the command to resend the same block, namely an ‘R’. The process will continue from step 2 after the block is received.
5. Given that the block arrived, and its checksum was calculated and verified, the server sends the next block of data.
6. The microcontroller resumes the protocol from step 2 until it successfully receives all blocks of the sample.
7. The microcontroller terminates the protocol by sending the command ‘X’.

### B. *Inter-Integrated Circuit (I2C) Protocol*

The project uses the I2C protocol for two purposes: to save and retrieve data blocks to and from the flash memory, and to send data to the digital to analog converter for output. This protocol uses two lines, a clock and a data line. The microcontroller operates in master mode, controlling the memory and the digital to analog converter. Occasionally, when reading data from flash memory, the microcontroller also operates in slave mode.

## IV. Materials Used

### A. *Microcontroller*

The microcontroller chosen for this project is the PIC18F47Q10 chip, produced by Microchip. This device was chosen primarily because of its clock rate and the fact that it has two I2C ports [1]. With these two ports, the reading from flash memory and the output to the digital to analog converter can be done in parallel, increasing efficiency. This device runs on a reduced instruction set processor [1].

### B. Flash Memory

The flash memory chip chosen for this project is the AT24CM02 chip, produced by Microchip. This device was chosen due to its I2C interface. Given that this device is produced as surface mount, an adapter was also purchased for using the product in a through-hole manner. This device is a two-megabit memory module, clocked at one megahertz [2].

This device is organized into 1024 pages of 256 bytes of data [2]. Due to this, and that the desired sample rate, 22,050 samples per second, requires double this amount for a decent sample length, two of these devices were purchased for use in the product.

### C. Digital to Analog Converter

The digital to analog converter chosen for this project is the DAC8571 chip, produced by Texas Instruments. This chip was chosen for the project due to it being a 16-bit converter, and that it uses an I2C interface to receive data.

Similar to the chosen flash memory, this device is only available as surface mount, so an adapter for this device was also ordered for use in a through-hole manner.

### D. Complete Materials List

The complete list of materials used for this project is as follows:

- Two 40-hole breadboards,
- One PIC18F47Q10 microcontroller,
- Two AT24CM02 flash memory chips,
- One DAC8571 digital to analog converter,
- One 64-megahertz crystal oscillator,
- Five ten-kilohm resistors
- Eight LED, for debugging purposes
- Eight 220-Ohm resistors, for debugging purposes
- Jumper Wires

## V. Data Flow

### A. Serial

All data in the serial protocol flows between the host computer and the microcontroller. This is done by using a custom-built utility, created in Java, to transmit the bytes contained in a sound sample file to the flash memory.

### B. I2C

All data in the I2C communication flows between the microcontroller and the digital to analog converter. The timing for I2C communication is precise and must occur in a specific sequence. The flash memory information is included here because it was initially used, but due to timing constraints and issues with getting it functioning, it was replaced with the microcontroller's internal program memory table functions.

All data writes to the flash memory are done in the page write style – where a complete page of 256 bytes is written by sending one address sequence, compared to single byte writes, where the address sequence would have to be transmitted before each data byte. The timing sequence for the page write to the flash memory can be found in figure 1 below.

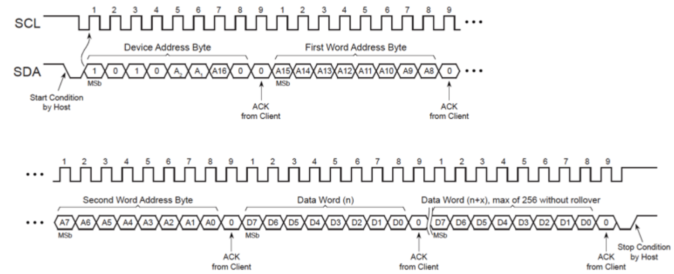


Fig. 1. Timing sequence for flash memory page write operations [2]

The sequence for writing to the flash memory in this way is as follows:

1. The microcontroller starts the I2C protocol by setting the data line to low.
2. The microcontroller sends the device address byte, with the read/write bit as a zero, formatted as in Table I.

TABLE I. Device Address Byte Format [2]

| Device Identifier | | | | Device Select | Address Most Significant Bits | | Read/Write Select |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | $A_2$ | A17 | A16 | RW |

3. The flash memory acknowledges the address byte by pulling the data line low for one byte time before releasing it.
4. The microcontroller sends the high bits of the address to be written to.
5. The flash memory acknowledges.
6. The microcontroller sends the low bits of the address.
7. The flash memory acknowledges.
8. The microcontroller sends an eight-bit data word to the flash memory.
9. The flash memory acknowledges.
10. The send-acknowledge process as described in steps eight and nine continue until all data words have been transmitted. At that time, the microcontroller stops the I2C protocol by pulling the data line high.

Similarly, for reading from the flash memory, we use a sequential random read technique. The timing sequence for the random and sequential reads can be found in figures 2 and 3 respectively.

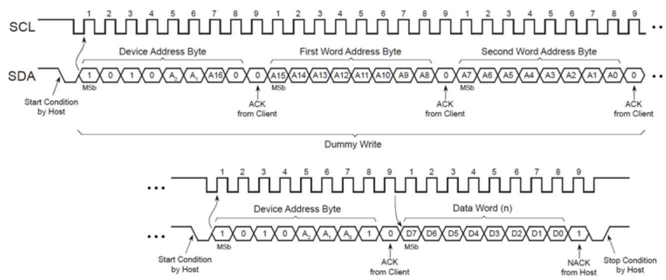


Fig. 2. Timing sequence for flash memory random read operations [2]

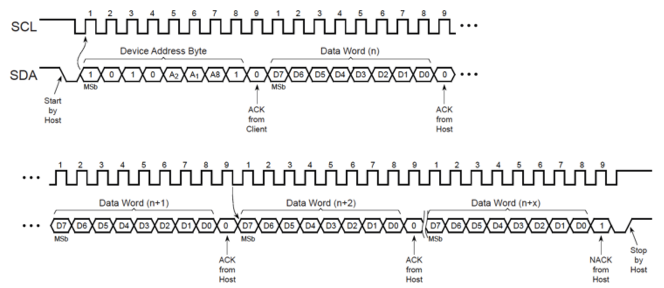


Fig. 3. Timing sequence for flash memory sequential read operations [2]

The sequence for reading from the flash memory is described below.

1. Steps one through seven of the previous sequence also apply in this case, because the random read first needs to set up a "dummy write" to load the requested address into the flash memory.
2. The microcontroller sends another start condition, as described in step one above.
3. The microcontroller sends the device address byte again, but with the read/write bit as a one instead of a zero.
4. The flash memory acknowledges.
5. The flash memory clocks in a data word and increments the address.
6. The microcontroller acknowledges in the same manner as the flash memory.
7. The flash memory repeats steps five and six until the microcontroller sends a non-acknowledgement by pulling the data line high (instead of low) for one bit time.
8. The microcontroller stops the I2C protocol as above.

### C. *Digital to Analog Converter*

The sequence to writing data to the digital to analog converter is as follows:

1. The microcontroller reads a data page from flash memory.
2. The microcontroller initiates an I2C protocol with the digital to analog converter by sending the device address byte, as formatted in table II below. The complete sequence can be shown in table III at the end of this section.

TABLE II. DEVICE ADDRESS BYTE FORMAT [3]

| Device Identifier | | | | Device Select | | | Read/Write Select |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | A0 | 0 | RW |

3. The microcontroller sends a control byte of all zeroes to the digital to analog converter.
4. The microcontroller sends the most significant byte of the data to be converted to the digital to analog converter.
5. The digital to analog converter acknowledges.
6. The microcontroller sends the least significant byte to the digital to analog converter.
7. The digital to analog converter acknowledges.
8. The sequence continues from step four to seven until the data page is read in to the digital to analog converter.
9. The sequence repeats from step one, excluding steps two and three, until all data is written to the digital to analog converter.
10. The microcontroller stops the I2C protocol with the digital to analog converter.

TABLE III. I2C SEQUENCE – DIGITAL TO ANALOG CONVERTER [3]

| Transmitter | Data | | | | | | | | Comment |
|---|---|---|---|---|---|---|---|---|---|
| Master | Start Condition | | | | | | | | Begin Sequence |
| Master | 1 | 0 | 0 | 1 | 1 | A0 | 0 | RW | Address |
| Slave | Acknowledge | | | | | | | | |
| Master | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Control |
| Slave | Acknowledge | | | | | | | | |
| Master | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | Data MSB |
| Slave | Acknowledge | | | | | | | | |
| Master | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Data LSB |
| Slave | Acknowledge | | | | | | | | |
| Master | Stop Condition, or Data MSB and LSB | | | | | | | | End Sequence |

## VI. PROGRAMMING

### A. *Microcontroller*

The microcontroller was programmed using Microchip's MPLAB IDE and the MPASM assembler. This assembler has a reduced instruction set consisting of 75 instructions [1].

### B. *File Transmission Software Package*

The file transmission utility was created in Java with the IntelliJ IDEA Community Edition IDE. It was created using JavaFX and a library for serial transmission, namely JSSC.

## VII. Challenges Faced

### A. *Learning a New Assembler*

The microcontroller assembler is similar to the assembler used in x86 systems taught in our Assembly Language Programming course, however, the instruction set, and syntax elements are different. The assembler used is a reduced instruction set on eight-bit data, so it is more difficult to seamlessly access more than that many bits of data. Also, there are multiple banks of registers on the microcontroller, and if a bank is being used outside of its bank, the program would not work. This process was difficult, but eventually I begam confident enough to begin writing code in this assembler for this project.

### B. *Serial Port Programming*

Being something that I've never done before, there were some issues with getting the serial port programming running on both the microcontroller and the Java based software package.

The microcontroller code was the hardest issue as it didn't work at the start. When it did eventually work, the task came to translate it into interrupt driven serial port programming.

The Java serial code was slightly easier, as there is a library available that was used in the software – and it has very good documentation which I was able to use to get it going.

### C. *Interrupt-Driven Programming*

Similar to the serial port programming previously discussed, this is also something that I've never had to do before. Getting the interrupts enabled was the first issue. Once those were sorted out, getting the serial port to generate interrupts and servicing them on the microcontroller was the main issue.

### D. *I2C Programming with One Port*

I2C is also something completely new to me. The main issue was getting the I2C port configured properly. Once that was done, the issue became where to place start and stop conditions and how to send data out through the port.

### E. *Combining Both I2C Ports*

Because one device type is on each I2C port, they need to work in parallel. This was the main issue, and there were two ways of implementing this – creating artificial time-slice threads on the microcontroller or running one port slightly faster than the other one for them to work better in parallel.

Due to the fact that the microcontroller does not natively support multithreading, we would have to create the threads from scratch – which seemed out of scope for this project.

### F. *Flash Memory Access Issues*

The issue faced with respect to the flash memory came near the end of the project. We noticed that the flash memory was only being written to on the even-numbered pages (e.g., page 0, page 2, page 4, etc.) leaving the odd-numbered pages (e.g., page 1, page 3, page 5, etc.) empty or filled with random values. This is not what was meant to happen, and due to the remaining time constraints, this was not able to be resolved in the preferred way.

### G. *Issues Writing to Microcontroller Program Memory*

Writing to the microcontroller's internal program memory was not as simple as expected. Using the datasheet didn't really help with it because it was unclear what parts of the example code were required to be implemented and which could be left out.

## VIII. Overcoming Challenges

### A. *Learning a New Assembler*

To overcome the challenges with the assembler, the microcontroller datasheet was a valuable resource. Contained in the datasheet is the complete instruction set summary, with individual pages on each instruction, listing the order of operands to the instruction and what they mean. Many instructions have multiple versions, so the datasheet was very good in showing how to write instructions.

### B. *Serial Port Programming*

The serial port programming issues were resolved by looking at example microcontroller code and other online resources [4,5] to assist in getting it working.

At one point in the project, the serial communication stopped, but that issue was fixed once it was realized that the serial port reception interrupt was somehow disabled, or not enabled at all.

### C. *Interrupt-Driven Programming*

The interrupt-driven programming challenges were solved with extensive review of the interrupt diagram shown in the datasheet, shown in figure X below. Aside from this, more sample code was found online.

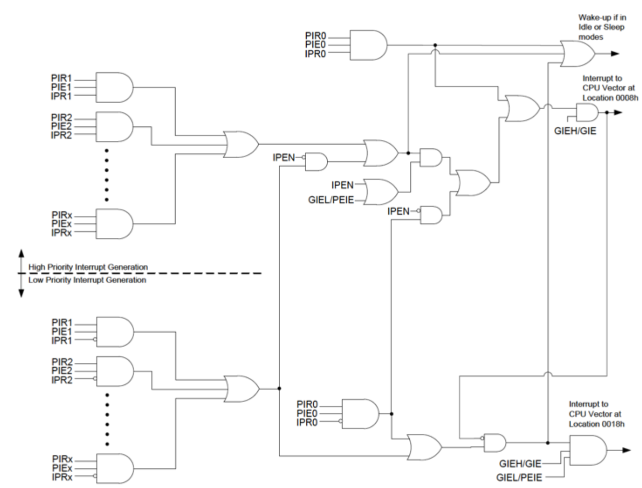


Fig. 4. Interrupt Circuit Diagram [1]

*D. I2C Programming with One Port*

Similar to the interrupt-driven programming, the issues with getting one I2C port working were resolved by looking at the supplied datasheet as well as sample code written by Microchip [6] for the purpose of introducing programmers to the I2C protocol in the MPASM language.

*E. Combining Both I2C Ports*

At one point in the project, all communications stopped, and the program would get stuck at one specific spot. This was resolved by reseating all of the pull-up resistors on the I2C clock and data lines. One had come out of one end, and this is what caused the issue. The ports require a constant logic 1 to be idle, and because the pull-up resistor came out, it was not connected to power or ground, and was considered floating.

*F. Flash Memory Access Issues*

As stated previously, this was resolved by using the microcontroller's program memory table commands. The program memory is a non-volatile memory, such like the flash memory that was intended to be used.

This makes the microcontroller code simpler because instead of accessing the I2C bus and dealing with that protocol, the data is stored in the program memory. Data is accessed and changed in the program memory by using the table pointer (TBLPTR) which is a 24-bit address locator that points to the location, in program memory, to access. Data is accessed using the table read command (TBLRD*+) which reads the data at the current TBLPTR address, loads it into memory and post-increments the TBLPTR registers. Similarly, it is written by using the table write command (TBLWT*+).

It also makes the program longer, as address spaces from 0x600 to approximately 0xFA0 are held by the sample data.

*G. Issues Writing to Microcontroller Program Memory*

The issues with writing to the microcontroller program memory were resolved by much trial-and-error executions of the code. The program memory has to be written to in the following sequence [1]:

1. Disable interrupts – The process must be repeated from the start if it is interrupted.
2. Initialize the memory address registers NVMADRU, NVMADRH and NVMADRL and the table pointer registers TBLPTRU, TBLPTRH and TBLPTRL.
3. Enable access to the non-volatile memory (NVM)
4. Perform a sector read unlock sequence
5. Perform a sector read
6. Perform a sector erase unlock sequence
7. Perform a sector erase. This fills the entire sector with 0xFF as modifying 1 bits to a 0 is permitted, but modifying a 0 bit to a 1 bit is not
8. Move all bytes to the Table Latch Holding Registers
9. Perform a sector write. This copies the value of all table latch holding registers to the program memory sector.
10. Disable access to the NVM
11. Enable interrupts, if they were enabled before the write process began.

## IX. Progress Achieved

To this point, I have the java-based PC application sending a file to the PIC chip. The PC sends the file bytes to the microcontroller over serial communication. The new sample is then able to be played as normal. The DAC functions have been implemented and were tested as working.

## X. Future Work

Future work I intend to complete on this project include:

12. Addition of a pressure switch that can be mounted to the interior of a hollow plastic cylinder, such to represent the chime, where it is hit with a small plastic mallet to set off the device
13. Addition of a dampening pedal, or button, to turn off the sound when pressed
14. Extrapolating the device to support multiple chime noises
15. Implementation of a feature of the device whereby if another chime is struck before the first is completed, the device will slowly dampen the first and begin playing the second
16. Implementation of the flash memory modules to store the sound sample as intended

## Acknowledgment

I would like to express my deepest appreciation to my project supervisor, Dr. George Townsend, for all of his support, guidance and wisdom in completing this project. There have been times this term where I began to regret choosing to do this project, but your encouragement during the term is what kept renewing my faith in myself, and what drove me to complete this project. I sincerely thank you for all you have done to help me this term.

## References

[1] Microchip, "28/40/44-pin, Low-Power, High-Performance Microcontrollers", PIC18F47Q10 datasheet, Jul. 2018 [Revised Oct. 2019]

[2] Microchip, "I²C-Compatible (Two-Wire) Serial EEPROM 2‑Mbit (262,144 x 8)", AT24CM02 datasheet, May 2019 [Revised Dec. 2020]

[3] Texas Instruments, "16-Bit, Low Power, Voltage Output, I2C Interface Digital-to-Analog Converters", DAC8571 datasheet, Dec. 2002 [Revised Jul. 2003]

[4] WikiPedia, "Serial Communication", https://en.wikipedia.org/wiki/Serial_communication#:~:text=In%20telecommunication%20and%20data%20transmission,link%20with%20several%20parallel%20channels, March 2021

[5] Circuit Digest, "Serial Communication Protocols", https://circuitdigest.com/tutorial/serial-communication-protocols, April 2019

[6] Microchip, "I2C Master Mode" PDF, http://ww1.microchip.com/downloads/en/devicedoc/i2c.pdf 2001

## APPENDIX A: FILE TRANSMISSION UTILITY CODE LISTING

```
package net.johnnyconsole.project.serial;


import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import jssc.SerialPort;
import jssc.SerialPortException;
import jssc.SerialPortList;


import java.io.File;
import java.io.IOException;
import java.nio.file.Files;


/**
 * @author Johnny Console
 * Course: COSC 4086 - Fourth Year Project
 * SerialFileTransmit: Transmit file bytes
 * over serial connection to load a new
 * sound sample into project memory
 */
public class SerialFileTransmit extends Application {


    private String portName;
    private final byte[] block = new byte[256];
    private int place = 0;
```

```
private File file;
private Button selectFile, beginTX;
private ComboBox<String> ports;
private Label status;
@Override
public void start(Stage ps) {
    //Root pane setup
    GridPane pane = new GridPane();
    pane.setPadding(new Insets(20));
    pane.setHgap(20);
    pane.setVgap(20);

    //UI object definition
    String[] list = SerialPortList.getPortNames();
    ports = new ComboBox<>(FXCollections.observableArrayList(list));
    selectFile = new Button("Select File...");
    beginTX = new Button("Begin Transmission");
    status = new Label("Select A File");

    //Attach properties to UI elements
    ports.getSelectionModel().select(0);
    ports.setMaxWidth(Double.MAX_VALUE);
    selectFile.setMaxWidth(Double.MAX_VALUE);
    beginTX.setDisable(true);
    beginTX.setMaxWidth(Double.MAX_VALUE);
    GridPane.setHalignment(status, HPos.CENTER);

    //Attach actions to UI elements
    selectFile.setOnAction(e -> selectFile(ps));
    beginTX.setOnAction(e -> beginTX(ps));

    //Add UI elements to root pane
    pane.add(status, 0, 0, 2, 1);
    pane.addRow(1, new Label("Select Serial Port:"), ports);
```

```
    pane.add(selectFile, 0, 2, 2, 1);
    pane.add(beginTX, 0, 3, 2, 1);

    //Stage and scene setup
    Scene scene = new Scene(pane);
    ps.setScene(scene);
    ps.setTitle("Transmit");
    ps.show();
}

private void selectFile(Stage ps) {
    //Get the File
    FileChooser chooser = new FileChooser();
    chooser.setTitle("Choose File to Transmit");
    chooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("Binary Sound Samples (*.raw)", "*.raw"));
    file = chooser.showOpenDialog(ps);
    if(file != null) {
        //If a file is selected, disable the select button ad enable the transmit button
        if(file.length() % 256 == 0) {
            beginTX.setDisable(false);
            selectFile.setDisable(true);
            selectFile.setText(file.getName());
            status.setText("File Selected - Ready");
        }
        else {
            status.setText("Invalid File - Length Invalid");
            System.out.println(file.length());
        }
    }
}
```

```
private void beginTX(Stage ps) {
    //Get the port information
    portName = ports.getSelectionModel().getSelectedItem();
    ports.setDisable(true);
    beginTX.setText("Transmitting Data...");
    beginTX.setDisable(true);
    //Start acting like a serial server
    serialServer(ps);
}

private void serialServer(Stage ps) {
    try {
        byte[] fileBytes = Files.readAllBytes(file.toPath());
        SerialPort port = new SerialPort(portName);
        if(port.openPort() && port.setParams(115200, 8, 1, 0)) {
            port.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);
            System.arraycopy(fileBytes, 0, block, 0, block.length);
            port.writeBytes(block);
            System.out.println("Waiting...");
            char command = (char)(port.readBytes(1)[0]);
            System.out.println("Command Received: " + command);
            while (command != 'X') {
                switch(command) {
                    case 'C':
                        int check = 0;
                        for (int i = 0;i < block.length; i++) {
                            if (i % 2 == 0) {
                                block[i] = (byte) (block[i] ^ 0x80);
                            }
                            check += (block[i] >= 0 ? block[i] : 256 + block[i]);
                            check %= 256;
                        }
```

```
                    port.writeByte((byte) (check % 256));
                    command = (char)(port.readBytes(1)[0]);
                    System.out.println("Command Received in C: " + command);

                    break;
                case 'N':
                    place += 256;
                    if (place == file.length()) {
                        System.out.println("End Of File");
                        command = 'X';
                        break;
                    } else {
                        System.arraycopy(fileBytes, place, block, 0, block.length);
                    }
                case 'R':
                    System.out.println("Transmitting Block " + ((place / 256) + 1));
                    port.writeBytes(block);
                    command = (char)(port.readBytes(1)[0]);
                    System.out.println("Command Received in R/N: " + command);
            }
        }
        ps.close();
    }
  } catch(SerialPortException | IOException ex) {
      System.err.println(ex.getMessage());
  }
 }
}
```

```
    INCLUDE <p18f47q10.inc>
  ;Serial TX pin = RC6
  ;Serial RX pin = RC7

TRIG MACRO
  COMF    LATD,f,ACCESS
  ENDM

;<editor-fold defaultstate="collapsed" desc="I2C1 Slave Macros">
SendReg1 MACRO FileReg
  MOVFF FileReg, WREG
  CALL I2C1Put
  CALL I2C1WaitIdle
  ENDM
Send1 MACRO SendB
  MOVLW   SendB
  CALL    I2C1Put
  CALL    I2C1WaitIdle
  ENDM

SendW1 MACRO
  CALL    I2C1Put
  CALL    I2C1WaitIdle
  ENDM

Recv1 MACRO
  I2C1RecEnable
  CALL I2C1WaitData
  MOVF SSP1BUF,w,ACCESS
  ENDM

I2C1Start MACRO
  BSF SSP1CON2,SEN_SSP1CON2,ACCESS
  ENDM

I2C1Restart MACRO
  BSF SSP1CON2,RSEN,ACCESS
  ENDM

I2C1Stop MACRO
  BSF SSP1CON2,PEN,ACCESS
  ENDM
I2C1ACK MACRO
  BCF SSP1CON2,ACKDT,ACCESS  ; acknowledge bit state to send
  BSF SSP1CON2,ACKEN,ACCESS  ; initiate acknowledge sequence
  BTFSC SSP1CON2,ACKEN,ACCESS  ; ack cycle complete??
  GOTO $-2 ; no, so loop again
  ENDM

I2C1NACK MACRO
  BSF SSP1CON2,ACKDT,ACCESS  ; acknowledge bit state to send
  BSF SSP1CON2,ACKEN,ACCESS  ; initiate acknowledge sequence
  BTFSC SSP1CON2,ACKEN,ACCESS  ; ack cycle complete??
  GOTO $-2 ; no, so loop again
  ENDM
```

```
I2C1RecEnable MACRO
   BSF SSP1CON2,RCEN,ACCESS
   ENDM
I2C1Disable MACRO
   BCF SSP1CON1,SSPEN,ACCESS
   ENDM

I2C1Get MACRO
   MOVF SSP1BUF,w,ACCESS
   ENDM
   ;</editor-fold>
;<editor-fold defaultstate="collapsed" desc="I2C2 Slave Macros">
SendReg2 MACRO FileReg
   MOVFF  FileReg,WREG
   CALL I2C2Put
   CALL I2C2WaitIdle
   ENDM

NBSendReg2 MACRO FileReg
   MOVFF  FileReg,WREG
   CALL I2C2Put
   ENDM
Send2 MACRO SendB
   MOVLW   SendB
   CALL    I2C2Put
   CALL    I2C2WaitIdle
   ENDM

SendW2 MACRO
   CALL    I2C2Put
   CALL    I2C2WaitIdle
   ENDM

Recv2 MACRO
   I2C2RecEnable
   CALL I2C1WaitIdle
   CALL I2C1WaitData
   MOVFF SSP2BUF,WREG
   ENDM

I2C2Start MACRO
   MOVFF SSP2CON2,WREG
   BSF WREG,SEN_SSP2CON2,ACCESS
   MOVFF WREG,SSP2CON2
   ENDM

I2C2Restart MACRO
   MOVFF SSP2CON2,WREG
   BSF WREG,RSEN,ACCESS
   MOVFF WREG,SSP2CON2
   ENDM

I2C2Stop MACRO
   MOVFF SSP2CON2,WREG
   BSF WREG,PEN,ACCESS
   MOVFF WREG,SSP2CON2
   ENDM
```

```
I2C2ACK MACRO
    MOVFF SSP2CON2,WREG
    BCF WREG,ACKDT,ACCESS  ; acknowledge bit state to send
    BSF WREG,ACKEN,ACCESS  ; initiate acknowledge sequence
    MOVFF WREG,SSP2CON2
    MOVFF SSP2CON2,WREG
    BTFSC WREG,ACKEN,ACCESS  ; ack cycle complete??
    GOTO $-4 ; no, so loop again
    ENDM

I2C2NACK MACRO
    MOVFF SSP2CON2,WREG
    BSF WREG,ACKDT,ACCESS  ; acknowledge bit state to send
    BSF WREG,ACKEN,ACCESS  ; initiate acknowledge sequence
    MOVFF WREG,SSP2CON2
    MOVFF SSP2CON2,WREG
    BTFSC WREG,ACKEN,ACCESS  ; ack cycle complete??
    GOTO $-4 ; no, so loop again
    ENDM

I2C2RecEnable MACRO
    MOVFF SSP2CON2,WREG
    BSF WREG,RCEN,ACCESS
    MOVFF WREG,SSP2CON2
    ENDM

I2C2Disable MACRO
    MOVFF SSP2CON2,WREG
    BCF WREG,SSPEN,ACCESS
    MOVFF WREG,SSP2CON2
    ENDM

I2C2Get MACRO
    MOVFF SSP2BUF,WREG
    MOVF WREG,w,ACCESS
    MOVFF WREG,SSP2BUF
    ENDM
    ;</editor-fold>
;<editor-fold desc="Serial Macros" defaultstate="collapsed">
sendCommand MACRO command
   BTFSS TX1STA, TRMT, ACCESS
   GOTO $-2
   MOVLW command
   MOVWF TX1REG, ACCESS
   ENDM
;</editor-fold>
```

```
;<editor-fold desc="Constants" defaultstate="collapsed">
REGF EQU 1
REGW EQU 0
SDI1 EQU RC4
SCL1 EQU RC3
SDI2 EQU RB2
SCL2 EQU RB1
TX1 EQU RC6
RX1 EQU RC7
OTMR0H EQU 0xFD
OTMR0L EQU 0x60
CHECK EQU 0x100
;</editor-fold>
;<editor-fold defaultstate="collapsed" desc="Configuration Bits">
; CONFIG1L
    CONFIG  FEXTOSC = OFF
    CONFIG  RSTOSC = HFINTOSC_64MHZ

    ; CONFIG1H
    CONFIG  CLKOUTEN = OFF
    CONFIG  CSWEN = ON
    CONFIG  FCMEN = ON

    ; CONFIG2L
    CONFIG  MCLRE = EXTMCLR
    CONFIG  PWRTE = OFF
    CONFIG  LPBOREN = OFF
    CONFIG  BOREN = SBORDIS

    ; CONFIG2H
    CONFIG  BORV = VBOR_190
    CONFIG  ZCD = OFF
    CONFIG  PPS1WAY = OFF
    CONFIG  STVREN = ON
    CONFIG  XINST = OFF

    ; CONFIG3L
    CONFIG  WDTCPS = WDTCPS_31
    CONFIG  WDTE = OFF

    ; CONFIG3H
    CONFIG  WDTCWS = WDTCWS_7
    CONFIG  WDTCCS = SC

    ; CONFIG4L
    CONFIG  WRT0 = OFF
    CONFIG  WRT1 = OFF
    CONFIG  WRT2 = OFF
    CONFIG  WRT3 = OFF


    ; CONFIG4H
    CONFIG  WRTC = OFF
    CONFIG  WRTB = OFF
    CONFIG  WRTD = OFF
    CONFIG  SCANE = ON
    CONFIG  LVP = ON
```

```
    ;</editor-fold>
;<editor-fold desc="Data Constant Block 0x00" defaultstate="collapsed">
    CBLOCK 0x00
    flag2
    flag3
    parity
    stopReq
    val
    t
    D10msA
    D10msB
    isrptr
    counter
    bytectr
    blocksH
    blocksL
    inblkH
    inblkL
    inbytectr
    command
    tor1
    tor2
    tor3
    saveSTATUS
    saveW
    saveFSR0H
    saveFSR0L
    saveBSR
    keepBSR
    dacout
    dacstat
    chksum
    firstByte
    debounce
    KTMR0H
    KTMR0L
    intState
    flip
    ENDC
;</editor-fold>
;<editor-fold desc="Vectors" defaultstate="collapsed">
    ORG 0x00
    CALL Initialize
    GOTO mainline
    ORG 0x08
    GOTO ISR
    ORG 0x18
    GOTO ISR
;</editor-fold>
;<editor-fold desc="Initialization" defaultstate="collapsed">
Initialize
    CLRF isrptr
    ;Initialize Debug I/O Port
    MOVLB 0x0F
    MOVLW 0x0F
    MOVWF TRISA
    CLRF ANSELA
```

```
    CLRF LATA
    BSF LATA, RA4
    MOVWF WPUA
    CLRF TRISD
    CLRF LATD
    CLRF dacstat
    CLRF bytectr
    CLRF parity
    SETF stopReq
    CLRF inblkH
    CLRF inblkL
    CLRF debounce
    MOVLW OTMR0H
    MOVWF KTMR0H
    MOVLW OTMR0L
    MOVWF KTMR0L
;<editor-fold desc="Initialize EUSART1" defaultstate="collapsed">
    MOVLB 0x0E ;Set up PPS registers for EUSART1
    MOVLW 0x17
    MOVWF RX1PPS
    MOVLW 0x09
    MOVWF RC6PPS

    MOVLB 0x0F    ;Set up pins for EUSART1 TX/RX
    CLRF TRISC
    BSF TRISC, 7    ;EUSART1 RX
    BCF TRISC, 6    ;EUSART1 TX
    CLRF ANSELC
    MOVLW d'34'    ;Baud 115200
    MOVWF SP1BRGL
    CLRF SP1BRGH
    CLRF BAUD1CON
    BSF BAUD1CON, BRG16    ;16 bit baud generator
    MOVF RC1REG,W    ;Clear EUSART1 RX register
    BSF RC1STA,SPEN    ;Enable EUSART1
    BSF RC1STA,CREN    ;Enable EUSART1 RX
    BCF TX1STA, TX9    ;Select 8-bit TX
    BSF TX1STA, TXEN    ;Enable EUSART1 TX
    BCF TX1STA, SYNC_TX1STA ;Use Asynchronous Mode
    BCF TX1STA, BRGH    ;Use Low Baud Mode
;</editor-fold>
```

```
;<editor-fold desc="Initialize I2C Ports" defaultstate="collapsed">
    MOVLB 0x0E
    MOVLW 0x13
    MOVWF SSP1CLKPPS
    MOVLW 0x09
    MOVWF SSP2CLKPPS
    MOVLW 0x14
    MOVWF SSP1DATPPS
    MOVLW 0x0A
    MOVWF SSP2DATPPS
    MOVLW 0x0F
    MOVWF RC3PPS
    MOVLW 0x12
    MOVWF RB2PPS
    MOVLW 0x10
    MOVWF RC4PPS
    MOVLW 0x11
    MOVWF RB1PPS
    MOVLB 0x0F
    MOVLW 0xFF
    MOVWF TRISC
    MOVWF TRISB
    CLRF ANSELC
    CLRF ANSELB
    MOVLW b'00011000'
    MOVWF INLVLC
    CLRF SLRCONC
    MOVLW b'00000110'
    MOVWF INLVLB
    CLRF SLRCONB
    CALL softReset1
    CALL softReset2
    I2C1Disable
    I2C2Disable
    CALL D10mSec
    CALL D10mSec
    CALL I2C1Setup
    CALL I2C2Setup
    I2C1Start
    I2C2Start
    CALL I2C1WaitIdle
    I2C1Stop
    CALL I2C1WaitIdle
    CALL I2C2WaitIdle
    I2C2Stop
    CALL I2C2WaitIdle
;</editor-fold>
```

```
;<editor-fold desc="Initialize Interrupts" defaultstate="collapsed">
    MOVLB 0x0E
    BCF IPR3,RC1IP
    BCF PIE3,RC1IE
    BCF PIR0,TMR0IF
    BCF PIE0,TMR0IE
    MOVLW 0x90
    MOVWF T0CON0
    MOVLW 0x40
    MOVWF T0CON1
    BCF INTCON,IPEN
    BSF INTCON,PEIE_GIEL
    BSF INTCON,GIE_GIEH
    RETURN

;</editor-fold>

;</editor-fold>
;<editor-fold desc="Time Out Functions" defaultstate="collapsed">
TimeoutReset
    MOVLB 0x00
    CLRF tor1
    CLRF tor2
    CLRF tor3
    RETURN
TimeoutInc
    MOVLB 0x00
    INCF tor1
    BZ inc2
    RETURN
inc2:
    INCF tor2
    BZ inc3
    RETURN
inc3:
    INCF tor3
    RETURN
;</editor-fold>
;<editor-fold desc="Checksum Function" defaultstate="collapsed">
checksum
    MOVLW 0x00
    MOVWF counter
    LFSR FSR0, 0x100
    MOVLW 0x00

nextNum:
    ADDWF POSTINC0, REGW, ACCESS
    DECFSZ counter, REGF, ACCESS
    GOTO nextNum
    RETURN
;</editor-fold>
;<editor-fold defaultstate="collapsed" desc="I2C1 Master Functions">
```

```
I2C1WaitIdle
    MOVF SSP1CON2,w,ACCESS
    ANDLW 0x1f ;Any of these? SEN,PEN,RSEN,RCEN,ACKEN
    BTFSS STATUS,Z,ACCESS
    BRA I2C1WaitIdle
    BTFSC SSP1STAT,R_W,ACCESS  ; transmission in progress?
    BRA I2C1WaitIdle
    RETURN

I2C1WaitData
    BTFSS SSP1STAT,BF,ACCESS
    GOTO I2C1WaitData
    RETURN

I2C1Setup
    MOVLW 0x28 ; enable MSSP as master
    MOVWF SSP1CON1,ACCESS
    MOVLW 0x20
    MOVWF SSP1ADD,ACCESS
    RETURN

I2C1Put ;RETURN 0 is okay, otherwise -1
    MOVWF SSP1BUF,ACCESS
    BTFSS SSP1CON1,WCOL,ACCESS
    RETLW 0
    BCF SSP1CON1,WCOL,ACCESS  ; clear collision flag
    RETLW -1

I2C1GotAck ; RETURN 0 if okay, -1 otherwise
    BTFSC SSP1CON2,ACKSTAT,ACCESS
    RETLW -1
    RETLW 0
;</editor-fold>
;<editor-fold defaultstate="collapsed" desc="I2C2 Master Functions">

I2C2WaitIdle
    MOVFF SSP2CON2,WREG
    ANDLW 0x1f ;Any of these? SEN,PEN,RSEN,RCEN,ACKEN
    BTFSS STATUS,Z,ACCESS
    BRA I2C2WaitIdle
    MOVFF SSP2STAT,WREG
    BTFSC WREG,R_W,ACCESS  ; transmission in progress?
    BRA I2C2WaitIdle
    RETURN

I2C2WaitData
    MOVFF SSP2STAT,WREG
    BTFSS WREG,BF,ACCESS
    GOTO I2C2WaitData
    RETURN

I2C2Setup
    MOVLW 0x28 ; enable MSSP as master
    MOVFF WREG,SSP2CON1
    MOVLW 0x20
    MOVFF WREG,SSP2ADD
    RETURN
```

```
I2C2Put ;RETURN 0 is okay, otherwise -1
    MOVFF WREG,SSP2BUF
    MOVFF SSP2CON1,WREG
    BTFSS WREG,WCOL,ACCESS
    RETLW 0
    BCF WREG,WCOL,ACCESS  ; clear collision flag
    MOVFF WREG,SSP2CON1
    RETLW -1

I2C2GotAck ; RETURN 0 if okay, -1 otherwise
    MOVFF SSP2CON2,WREG
    BTFSC WREG,ACKSTAT,ACCESS
    RETLW -1
    RETLW 0
;</editor-fold>
;<editor-fold defaultstate="collapsed" desc="Helper Functions">
softReset1
    BSF     LATC,SDI1
    BSF     LATC,SCL1
    BCF     TRISC,SDI1
    BCF     TRISC,SCL1
    CALL    D1uSec
    BCF     LATC,SDI1
    CALL    D1uSec
    BCF     LATC,SCL1
    CALL    D1uSec
    CALL    D1uSec
    BSF     LATC,SDI1
    MOVLW   d'9'
resloop1:
    CALL    D1uSec
    BSF     LATC,SCL1
    CALL    D1uSec
    BCF     LATC,SCL1
    DECFSZ  WREG
    BRA     resloop1
    CALL    D1uSec
    CALL    D1uSec
    BCF     LATC,SDI1
    CALL    D1uSec
    BSF     LATC,SCL1
    CALL    D1uSec
    BSF     LATC,SDI1
    BSF     TRISC,SDI1
    BSF     TRISC,SCL1
    RETURN
```

```
softReset2
   BSF     LATB,SDI2
   BSF     LATB,SCL2
   BCF     TRISB,SDI2
   BCF     TRISB,SCL2
   CALL    D1uSec
   BCF     LATB,SDI2
   CALL    D1uSec
   BCF     LATB,SCL2
   CALL    D1uSec
   CALL    D1uSec
   BSF     LATB,SDI2
   MOVLW   d'9'
resloop2:
   CALL    D1uSec
   BSF     LATB,SCL2
   CALL    D1uSec
   BCF     LATB,SCL2
   DECFSZ  WREG
   BRA     resloop2
   CALL    D1uSec
   CALL    D1uSec
   BCF     LATB,SDI2
   CALL    D1uSec
   BSF     LATB,SCL2
   CALL    D1uSec
   BSF     LATB,SDI2
   BSF     TRISB,SDI2
   BSF     TRISB,SCL2
   RETURN
D1uSec
   NOP
   NOP
   NOP
   NOP
   RETURN


D20uSec
   MOVLW  4
   GOTO   delPatch
D1mSec
   MOVLW  .21
   GOTO   delPatch
D10mSec
   MOVLW   .210
delPatch
   CLRF    D10msA,ACCESS
   MOVWF   D10msB,ACCESS
rms:
   DECFSZ  D10msA,1,ACCESS
   GOTO    rms
   DECFSZ  D10msB,1,ACCESS
   GOTO    rms
   RETURN
```

```
INCNVMADDR
    BCF STATUS, C
    RLCF NVMADRL
    BTFSS STATUS, Z
    RETURN
    INCF NVMADRH
    BTFSS STATUS, Z
    RETURN
    INCF NVMADRU
    RETURN

initBlkWrt
    MOVLW 0x00
    MOVWF NVMADRU
    MOVLW 0x06
    MOVWF NVMADRH
    MOVLW 0x00
    MOVWF NVMADRL
    RETURN
;</editor-fold>
;<editor-fold desc="DAC Functions">
DACON
    MOVFF BSR, keepBSR
    BCF LATA, RA4
    BSF LATA, RA5
    CLRF stopReq
    call I2C2WaitIdle
    I2C2Start
    CALL I2C2WaitIdle
    Send2 0x98
    Send2 0x10
    MOVLB 0x0E
    MOVFF KTMR0H, TMR0H
    MOVFF KTMR0L, TMR0L
    BSF PIE0,TMR0IE
    MOVFF keepBSR, BSR
    RETURN

DACOFF
    MOVFF BSR, keepBSR
    CLRF debounce
    BSF LATA, RA4
    BCF LATA, RA5
    MOVLB 0x0E
    BCF PIE0,TMR0IE
    CALL I2C2WaitIdle
    I2C2Stop
    CALL I2C2WaitIdle
    I2C2Start
    CALL I2C2WaitIdle
    Send2 0x98
    Send2 0x10
    Send2 0x80
    Send2 0
    I2C2Stop
    MOVFF keepBSR, BSR
    RETURN
```

```
;</editor-fold>
 ;<editor-fold desc="Serial Functions">


nextBlkWrt:
; Code sequence to modify one complete sector of PFM
READ_BLOCK:
    MOVFF INTCON,intState
    BCF INTCON, GIE ; disable interrupts
    BSF NVMCON0, NVMEN ; enable NVM
   ; ----- Required Sequence -----
    MOVLW 0BBh
    MOVWF NVMCON2 ; first unlock byte = 0BBh
    MOVLW 44h
    MOVWF NVMCON2 ; second unlock byte = 44h
    BSF NVMCON1, SECRD ; start sector read (CPU stall)
   ; ------------------------------
ERASE_BLOCK: ; NVMADR is already pointing to target block
; ----- Required Sequence -----
    MOVLW 0CCh
    MOVWF NVMCON2 ; first unlock byte = 0CCh
    MOVLW 33h
    MOVWF NVMCON2 ; second unlock byte = 33h
    BSF NVMCON1, SECER ; start sector erase (CPU stall)
; ------------------------------
MODIFY_BLOCK:
    MOVFF NVMADRU,TBLPTRU
    MOVFF NVMADRH,TBLPTRH
    MOVFF NVMADRL,TBLPTRL
    LFSR FSR0, 0x100
    MOVLW 0x00
blkLoop:
    MOVWF flip
    MOVFF POSTINC0, WREG
    BTFSC FSR0L, 0
    XORLW 0x80
    MOVWF TABLAT
    MOVFF flip, WREG
    TBLWT*+
    DECFSZ WREG
    GOTO blkLoop
PROGRAM_MEMORY: ; NVMADR is already pointing to target block
    ; ----- Required Sequence -----
     MOVLW 0DDh
     MOVWF NVMCON2 ; first unlock byte = 0DDh
     MOVLW 22h
     MOVWF NVMCON2 ; second unlock byte = 22h
     BSF NVMCON1, SECWR ; start sector programming (CPU stall)
    ; ------------------------------
     BCF NVMCON0, NVMEN ; disable NVM
     BTFSC intState, GIE ; only re-enable interrupts if they were enabled
     BSF INTCON, GIE ; re-enable interrupts
     INCF NVMADRH
     BTFSS STATUS,Z
     RETURN
     INCF NVMADRU
     RETURN
```

```
downloadSample
   CLRF isrptr
   SETF stopReq
   MOVLB 0x0E
waitDAC:
   BTFSC PIE0, TMR0IE
   GOTO waitDAC
   CALL initBlkWrt
   MOVF RC1REG, w
   BSF PIE3, RC1IE
   LFSR FSR0, 0x100
   BCF LATA, RA4
   BCF LATA, RA5
   BSF LATA, RA7
waitSet:
   MOVF isrptr
   BTFSC STATUS, Z, ACCESS
   GOTO waitSet
waitZero:
   MOVF isrptr
   BTFSS STATUS, Z, ACCESS
   GOTO waitZero
   CALL checksum
   MOVWF chksum
   MOVFF CHECK, firstByte
   sendCommand 'C'
waitChk:
   MOVF isrptr
   BTFSC STATUS, Z, ACCESS
   GOTO waitChk
   MOVF chksum, w
   LFSR FSR0, 0x100
   CPFSEQ INDF0
   GOTO resend
   MOVFF firstByte, CHECK

   CALL nextBlkWrt

   CLRF isrptr
   MOVFF NVMADRU,WREG
   XORLW 0x01
   BTFSS STATUS, Z
   GOTO sendN
   MOVFF NVMADRH,WREG
   XORLW 0xFA
   BTFSC STATUS, Z
   GOTO sendX
sendN:
   sendCommand 'N'
   GOTO waitSet
sendX:
   sendCommand 'X'
   GOTO done
```

```
resend:
    CLRF isrptr
    sendCommand 'R'
    GOTO waitSet

done:
    BCF PIE3, RC1IE
    BCF LATA, RA7
    BSF LATA, RA4
    RETURN
 ;</editor-fold>

mainline:
    BTFSS PORTA, RA1
    CALL downloadSample
    BTFSS PORTA, RA2
    GOTO play
    BTFSS PORTA, RA3
    GOTO stop
    GOTO mainline

play:
    MOVF debounce, f
    BTFSS STATUS, Z
    GOTO mainline
    INCF debounce
    CALL DACON
    MOVLW 0x00
    MOVFF WREG, TBLPTRU
    MOVLW 0x06
    MOVFF WREG, TBLPTRH
    MOVLW 0x00
    MOVFF WREG, TBLPTRL
    GOTO mainline

stop:
    BCF LATA, RA5
    SETF stopReq
    GOTO mainline

ISR:
    MOVFF STATUS, saveSTATUS
    MOVFF WREG, saveW
    MOVFF BSR, saveBSR
    MOVFF FSR0H, saveFSR0H
    MOVFF FSR0L, saveFSR0L
    TRIG
    MOVLB 0x0E
    BTFSS PIE0, TMR0IE
    GOTO checkSerial
    BTFSS PIR0, TMR0IF
    GOTO checkSerial
    BCF PIR0, TMR0IF
    MOVFF KTMR0H, TMR0H
    MOVFF KTMR0L, TMR0L

    TBLRD*+
```

```
   NBSendReg2 TABLAT
   COMF parity
   BTFSS STATUS, Z
   GOTO doMore
   MOVF debounce, f
   BTFSS STATUS, Z
   INCF debounce
   MOVF stopReq
   BTFSS STATUS, Z
   GOTO off
doMore:
   MOVFF TBLPTRU,WREG
   XORLW 0x01
   BTFSS STATUS, Z
   GOTO checkSerial
   MOVFF TBLPTRH,WREG
   XORLW 0xFA
   BTFSC STATUS, Z
off:
   CALL DACOFF

checkSerial:
   BTFSS PIR3, RC1IF
   GOTO restore
   MOVLW 0x01
   MOVWF FSR0H
   MOVFF isrptr, FSR0L
   MOVLB 0x00
   MOVFF RC1REG, INDF0
   INCF isrptr
restore:
   MOVFF saveW, WREG
   MOVFF saveBSR, BSR
   MOVFF saveFSR0H, FSR0H
   MOVFF saveFSR0L, FSR0L
   MOVFF saveSTATUS, STATUS
   RETFIE

   org 0x600
   INCLUDE A3.txt          ;The bytes of the included sample file

   END
```

## APPENDIX C: SAMPLE PORTION OF THE A3.TXT INCLUDE FILE

The file is a text file that contains the define word (dw) instructions for each word of data in the sample. In this case, words are 16-bit data values.

The first ten lines of the file, at the start of the sample, look like this:

```
dw 0x1d7f
dw 0x357f
dw 0x407f
dw 0x287f
dw 0x3c7f
dw 0x3d7f
dw 0x5c7f
dw 0x0a80
dw 0x6d7f
dw 0x6a7f
```

The last ten lines of the file, at the end of the sample, look like this:

```
dw 0x0180
dw 0x0080
dw 0x0080
dw 0x0080
dw 0x0080
dw 0x0080
dw 0x0080
dw 0x0180
dw 0x0080
dw 0x0080
dw 0x0080
```

The file is very large as it contains every single word of the sample data. This is why only the first and last ten lines of the file are shown.

This file is used to load the sample that would be shipped with the system. When the user downloads a new sample, it overwrites the included sample.