PIC based MIDI synthesizer

Yann Vernier yannv@kth.se

May 2, 2010

Abstract

This document describes a simple MIDI synthesizer implemented in a single PIC microcontroller, developed for the IL131V Basic Digital Theory with PIC processor course at the Royal Institute of Technology in Sweden.

Contents

I Functional description 2

1 Main code 2

2 Interrupt code 2

II Implementation 2

3 Preamble 4

4 Declarations 5

5 Interrupt handler 6
5.0.1 Performance 9

6 Setup code 9

7 Main loop 11

8 Utility functions 12
8.1 Sound sample lookup 13
8.1.1 Performance 13
8.2 MIDI note translation 13
8.2.1 Performance 17
8.3 MIDI receive state machine 17

9 Constants 20

10 Build system 21

III Verification 22

11 Checklist **22**

List of Figures

1 Circuit diagram of synthesizer . . . 2
2 JSP overview of main program . . . 3
3 JSP overview of interrupt handler . . . 3

Part I
Functional description

The circuit (figure 1) is very simple. A standard MIDI signal is connected to the PIC RX pin using an optocoupler (for the prototype, a standard PC joystick port to MIDI adaptor). The PIC, clocked at 20MHz using an external crystal, processes the incoming signals and generates a PWM audio signal on the CCP1 pin, biased around half the supply voltage. Because this is a PWM type signal rather than a "1-bit DAC" style (with 1s and 0s evenly distributed), a low pass filter is added to reduce the 19kHz signal, which is audible to some people. This frequency could be raised by reducing the timer period, but the time is already rather tight for the interrupt routine. A possible solution is setting the interrupt to trigger less often using the postscaler.

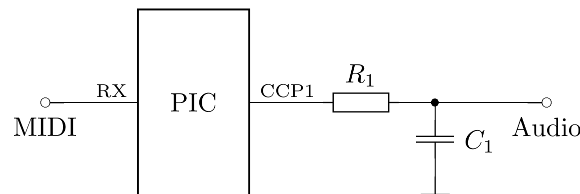

Figure 1: Circuit diagram of synthesizer

1 Main code

The main part of the program (figure 2) handles initialization and low priority processes. In this case, the initialization sets up I/O pins, UART, timer and PWM, and enables an interrupt for the high priority task of audio synthesizing. After all that is done, it only receives MIDI control signals, interprets them and reconfigures the synthesizer state to play notes.

The MIDI packet processing routine implements a simple state machine, watching for note on or off events. On finding such, it will make the corresponding changes to the counter steps in the signal generators.

2 Interrupt code

The high priority code (figure 3) runs in one interrupt handler. It has to run at a precise interval, conveniently defined by the timer. As such, while there is a conceptual loop, it is precisely bounded by the number of compiled in channels.

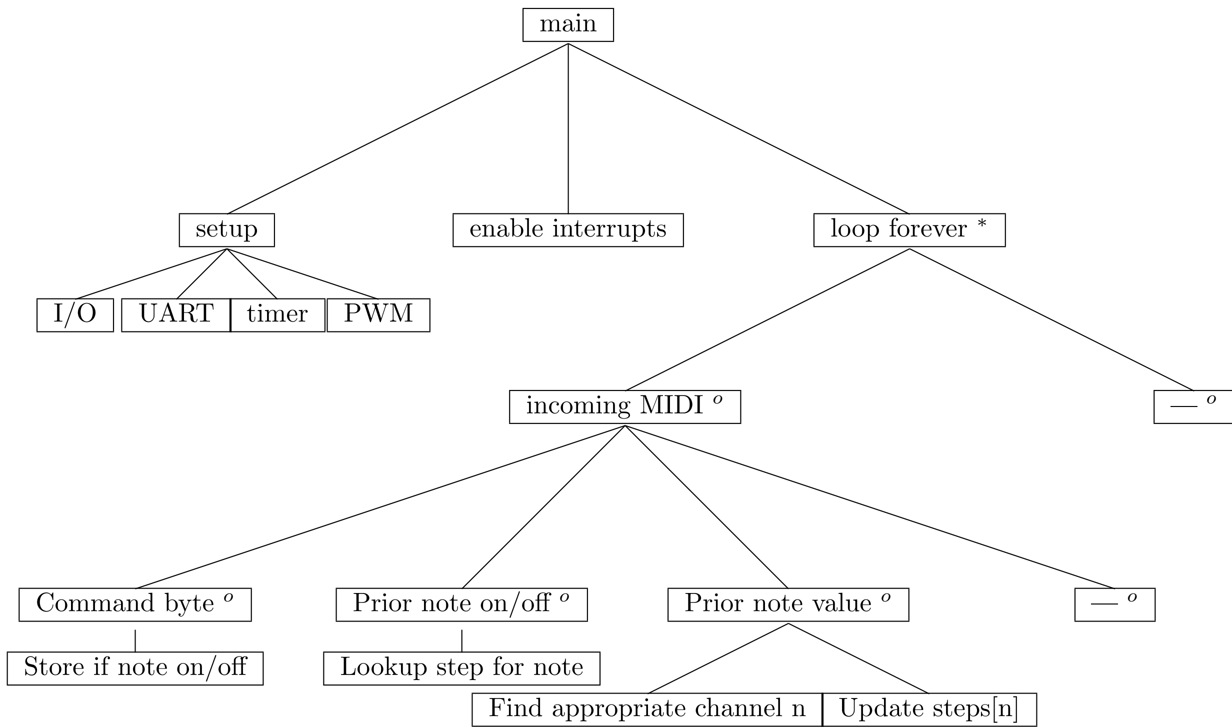

Figure 2: JSP overview of main program

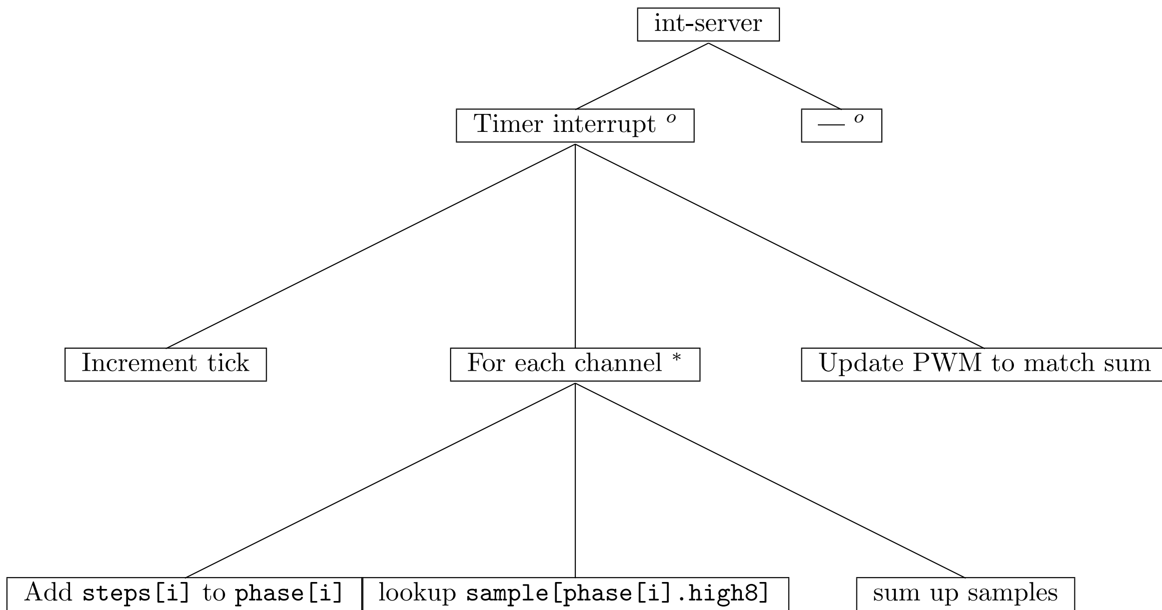

Figure 3: JSP overview of interrupt handler

Part II
Implementation

A wavetable structure was chosen, as this lends itself well to software mixing (using simple addition). The basic code structure needed, then, was some I/O setup, a main loop reading MIDI commands, and a perfectly

regular routine for updating the PWM value, which would be an average of signals. Those signals are calculated using table lookups, fed by counters running at the frequencies instructed via MIDI. The regular updates are simply achieved by using the interrupt from the same timer that drives the PWM.

These C sections are not individual files, but code fragments spread through this report. They are collected by Noweb, as per the following description, to generate a single module for CC5X to compile.

4a ⟨*synth.c* 4a⟩≡
⟨*preamble.c* 4b⟩
⟨*declarations.c* 5⟩
⟨*interrupt-handler.c* 6⟩
⟨*utilities* 12⟩
⟨*setup.c* 9⟩
⟨*main.c* 11⟩
⟨*constants.c* 20b⟩

This code is written to file `synth.c`.

3 Preamble

This section consists mainly of information for the compiler, choosing the PIC model, its configuration, and setting up some macros to make the remaining code more flexible.

To keep the interrupt routine fast, bank switching code is *disabled* inside it. This includes routines it calls, like the sound sample lookup. For this to work, all registers accessed by those routines must reside in bank 0.

4b ⟨*preamble.c* 4b⟩≡ (4a)

```
/* Knudsen CC5X program for a wavetable based MIDI synthesizer

                --------  --------
               |        \/        |
               |RA2   16F628   RA1|
               |RA3            RA0|
               |RA4-od    RA7/OSC1|----20 MHz Crystal
               |RA5/MCLR  RA6/OSC2|---/
        GND ---|Vss            Vdd|--- +5V
  lightdiode -<-|RB0/INT  (RB7)/PGD|
     MIDI in ->-|RB1/Rx   (RB6)/PGC|
               |RB2/Tx         RB5|
     speaker -<-|RB3/CCP  (RB4)/PGM|-1k- Gnd
               |__________________|
*/

#pragma chip PIC16F628A

#include <int16Cxx.h>

/* WDTE=off, FOSC=HS for crystal */
#pragma config WDTE=off, FOSC=HS
// To avoid unnecessary bank switching, we stick to one RAM bank
// This must be bank 0 as it holds TMR2 and RCREG
#pragma rambank 0
// Let the code know that 24-bit arithmetic is disabled
#define KNUDSEN_CRIPPLED 1
```

4 Declarations

In this section we find all global variables, including the state for the signal generators, MIDI command parser, wavetable locations and such. Declarations for functions used further down also go in this section.

5 ⟨*declarations.c* 5⟩≡ (4a) 17a ▷

```
// Output register for the LED
#pragma bit lightdiode @ PORTB.0

// Number of synthesizer channels, thus polyphonic voices, supported in this build
#define CHANNELS 8
// Whether to generate a test sequence of tones from the main loop
#define TEST_SEQ 0

// Note: convert return value to int, values in sine table are signed!
// W is high bits of flash address
char lookup(char entry, char W);
// Subroutine for converting midi notes into counter steps
// Argument in range 0-127 (valid MIDI notes), return value in step (below)
void midi2step(int x);

// Global variables
uns24 step;     // Return value from midi2step()

// Signal generator state
uns24 steps[CHANNELS], phase[CHANNELS];
char bank[CHANNELS];
#if TEST_SEQ
char tick;      // Used for synchronisation in main loop test code
#endif
```

5 Interrupt handler

The interrupt handler is run for every output sample, at a rate of $\frac{Fosc/4}{255} \approx 19.6\text{kHz}$. This leaves it not much more than 200 instruction cycles in which to run, during which is must process all channels configured. Therefore a closer inspection of the generated code, as well as careful optimization, is required.

The C preprocessor is used to generate an inline code section rather than a for loop, because there is much less overhead in constant addressing. The code is also complicated by the fact that Knudsen's free compiler does not allow 24 bit arithmetic.

6 ⟨*interrupt-handler.c* 6⟩≡ (4a)

```
/* The interrupt handler has to be early. And Knudsen's can't handle relocation. */
#pragma origin 4
interrupt int_server(void) {
        int_save_registers;
        char sv_PCLATH = PCLATH;
        // save two instructions here as this is the only enabled interrupt
        if (1 /*TMR2IF*/) {
                TMR2IF=0;       // Clear interrupt flag, allowing interrupt to repeat
#if TEST_SEQ
                tick++;         // Inform main loop that time has passed
#endif
                // unsigned to reduce wasted instructions in final division (shift)
                unsigned long level=0;

                // Advance each channel's phase one step and collect signal levels
                // I don't know how to get Knudsen to unroll a for loop

                // Method for extending addition found at:
                // http://www.piclist.com/techref/microchip/math/add/24b.htm

#define PHASECOUNT(NUM)                                                         \
        #if CHANNELS>NUM                                                        \
        #if KNUDSEN_CRIPPLED                                                    \
                phase[NUM].low8+=steps[NUM].low8;                               \
                W=steps[NUM].mid8;                                              \
                if (Carry==1) W++;                                              \
                phase[NUM].mid8+=W;                                             \
                W=steps[NUM].high8;                                             \
                if (Carry==1) W++;      /* Extend the addition to 24 bits */    \
                phase[NUM].high8+=W;                                            \
        #else                                                                   \
                phase[NUM]+=steps[NUM];                                         \
        #endif                                                                  \
                level+=lookup(phase[NUM].high8, bank[NUM]);                     \
        #endif

                // This section is performance critical
                // The only function called here, lookup, does not affect bank selection
#pragma updateBank 0
                PHASECOUNT(0)
                PHASECOUNT(1)
                PHASECOUNT(2)
```

```
                PHASECOUNT(3)
                PHASECOUNT(4)
                PHASECOUNT(5)
                PHASECOUNT(6)
                PHASECOUNT(7)

                // Finally, compute an average from the signal sum
#if CHANNELS==1
#elif CHANNELS==2
                CCP1X=0;
                CCP1X|=level.0;
#elif CHANNELS==3
#elif CHANNELS==4
                CCP1X=0;
                CCP1X|=level.1;
                CCP1Y=0;
                CCP1Y|=level.0;
#elif CHANNELS==8
                CCP1X=0;
                CCP1X|=level.2;
                CCP1Y=0;
                CCP1Y|=level.1;
#elif CHANNELS==16
                CCP1X=0;
                CCP1X|=level.3;
                CCP1Y=0;
                CCP1Y|=level.2;
#else
#error Fix the code for more channels..
#endif

                // As the MSBs are ignored, we can optimize powers of 2 well
#if CHANNELS==2
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
#elif CHANNELS==4
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
#elif CHANNELS==8
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
                level.high8=rr(level.high8);
                level.low8=rr(level.low8);
#else
                level/=CHANNELS;
#endif
                CCPR1L=level.low8;
```

```
        }
        PCLATH = sv_PCLATH;
        int_restore_registers;
}
// Enable bank switching in other code, where time is less critical
#pragma updateBank 1
```

5.0.1 Performance

Once in the interrupt routine, 15 cycles are spent just saving, setting and restoring state (STATUS, PCLATH and such). Each channel requires 24 cycles, of which 6 are spent in the sample lookup call. Then the sum is converted to store in the PWM registers, which takes another 14 cycles.

Thus, the interrupt routine uses (for 8 channels) 221 cycles. The period is 255 cycles, so only about 30 per sample period (or 180 per possible MIDI byte) are left for the main loop.

6 Setup code

The setup code is used to configure the I/O devices, including setting up the PWM mode, MIDI baud rate and reception, and timer 2 period. It is fairly straightforward code, with the only notable detail that it should leave the bank selection as 0 on exit, because it is the only function to use any other value.

9 ⟨*setup.c* 9⟩≡ (4a)

```
// Not only does Knudsen not manage to inline the single use function,
// it fails to even compile if it's "static inline".
inline static void setup(void) {
#pragma updateBank exit=0
        int i;

        TRISB = 0b11110010;     // Pins 0 (LED), 2 (TX) and 3 (PWM) output
        PR2 = 0xfe;     /* PWM timer period */
        /* 0b00.xx.xxxx         unused
         * 0bxx.00.xxxx         PWM least significant bits
         * 0bxx.xx.11xx         PWM mode        */
        CCP1CON = 0b00.00.1111;
        /* 0b0.xxxx.x.xx        unused
         * 0bx.0000.x.xx        postscale 1:1
         * 0bx.xxxx.1.xx        timer 2 on
         * 0bx.xxxx.x.00        prescale 1:1    */
        T2CON=0b0.0000.1.00;
        /* Reset interrupt flags */
        PIR1 = 0;
        /* Enable timer 2 interrupt */
        PIE1=0b0000.0010;

        /* UART setup for MIDI */
        /* spbrg=fosc/(64*baud)-1=9 for 20MHz crystal, 31.25kbaud MIDI */
        SPBRG=9;
        /* RCSTA=0b1xxx.xxxx  serial port enable
                   x0xx.xxxx  8 bit mode
                   xxXx.xxxx  single receive (not used in asynch mode)
                   xxx1.xxxx  continuous receive enabled
                   xxxx.0xxx  address detection disabled
                   xxxx.xXxx  framing error
                   xxxx.xxXx  overrun error
                   xxxx.xxxX  9th/parity bit (unused in MIDI) */
        RCSTA=0b1001.0000;

        ⟨setup-midi-parser.c 17b⟩
}
```

7 Main loop

Besides calling the setup routine, the main loop has to set up new tones for the signal generator to play. This is done either by a built in simple sequencer or by listening for external MIDI commands, depending on the TEST_SEQ setting in the declarations.

11 ⟨*main.c* 11⟩≡ (4a)

```
// Main routine needs access to a few other registers for setup
void main(void)
{
#if TEST_SEQ
        long count=0;
        int j=0x45, i, n;
#endif

        lightdiode = 1;         // Show that program has started
        setup();
        /* enable interrupts from peripherals */
        INTCON=0b11000000;
        // Past this point, we stick to bank 0.
#pragma updateBank 0
        while(1) {
                // Handle incoming MIDI data
                ⟨uart-receive.c 18a⟩
#if TEST_SEQ
                if(tick-i>=0) {         // synchronize time
                        i+=20;
                        count++;
                        count&=0x3ff;
                        if(count==0) {
                                /* chord:
                                 *  c# d#   f# g# a#
                                 * c  d  e f  g  a  b
                                 * 0     4    7         */
                                n+=1;
                                n&=3;
                                switch(n) {
                                        case 0:
                                                midi2step(j);
                                                GIE=0;
                                                steps[0]=step;
                                                GIE=1;
                                                break;
        #if CHANNELS >= 2
                                        case 1:
                                                midi2step(j+4);
                                                GIE=0;
                                                steps[1]=step;
                                                GIE=1;
                                                break;
        #endif
        #if CHANNELS >= 3
```

```
                                        case 2:
                                                midi2step(j+7);
                                                GIE=0;
                                                steps[2]=step;
                                                GIE=1;
                                                break;
        #endif
                                        default:
                                                GIE=0;
                                                steps[0]=0;
        #if CHANNELS >=2
                                                steps[1]=0;
        #endif
        #if CHANNELS >=3
                                                steps[2]=0;
        #endif
                                                GIE=1;
                                                break;
                                }
                                if (j<0x45+2*12)
                                        j+=2;
                                else
                                        j=0x45-2*12;
                        }
                }
#endif // TEST_SEQ
        }
}
```

8 Utility functions

The utility functions are responsible for various translation steps. Some code, such as `midi2step`, must be early in memory.

12 ⟨*utilities* 12⟩≡ (4a)
⟨*lookup.c* 13a⟩
⟨*midi2step* 13b⟩
⟨*processmidi.c* 19⟩
⟨*resetmidi.c* 18b⟩

8.1 Sound sample lookup

The lookup routine is meant for extremely fast constant table lookups. Unfortunately, the PIC model chosen does not have a function to read directly from program space; therefore, calculated jumps to `RETLW` instructions are used. This means the lookup routine itself cannot be inlined, as a `CALL` is necessary to put the return address on the stack. The routine thus works much the same as Knudsen's automatically generated `_const` lookup functions, but assumes the array is 256 bytes aligned at a 256 byte boundary. These restrictions allow it to be 3 instructions in length rather than 12.

Ideally, Knudsen would have automatically done all this when a const array of 256 bytes was declared, but that would require relocation support - the lookup routine must not be adjacent to the table. Technically it could, but then the page below could not be used the same way. I tried doing this using alignment pragmas but Knudsen did not optimize for the case, and in fact altered the lookup function size such that a single pragma could *not* align the table.

13a ⟨*lookup.c* 13a⟩≡ (12)

```
char lookup(char entry, char W) {
    // Does not alter RAM bank selection
    #pragma updateBank entry=0
    #pragma updateBank exit=0
    PCLATH = W;   // Load the code bank number
    PCL = entry;  // two instructions, far jump to data array
    return 0;     // never reached
}
```

The extra argument `W` is used to select which 256-byte boundary the array starts from. This allows the same lookup routine to be used for different sounds. It's not a large difference in function size or speed because `PCLATH` will have to be loaded in all cases.

The return type of `char` is chosen because the `RETLW` instructions mean we must return in `W`; Knudsen returns `int` in a file register instead. Our actual signals are signed, but we compensate by an initial offset in `level` and a matching bias in the tables, as the PIC can more easily handle unsigned addition of different sized values.

8.1.1 Performance

The routine itself is quite minimized; store `PCLATH`, load and store `entry`, which branches off to the `RETLW` instruction containing data. This requires 6 cycles; the call takes a few as well.

8.2 MIDI note translation

The MIDI to step function converts MIDI tone numbers, in the range from 0 to 127, to their respective steps, precalculated using Python. Unlike most other modules, this is a separate file as it can also be compiled as a separate diagnostic program.

13b ⟨*midi2step* 13b⟩≡ (12)

```
#include "midi2step.c"
```

14 ⟨*midi2step.c* 14⟩≡

```
/* Frequency basis:
 *  crystal is 20MHz, PIC cycles are 20/4MHz=5MHz
 *  PWM period is 255 PIC cycles = 5/255=19.608kHz sample rate
 * highest MIDI tone is 127=58 haltones over middle A (440Hz)
 * lowest MIDI tone is 0=69 halftones below middle A = 5 octaves and 9 halftones
 * that's about 8Hz (8.175798915643707)
 * translates to 2398.2785461781523 samples per period

  Calculations (Python):
      crystal=20e6
      cpufreq=crystal/5
      samplerate=cpufreq/255
      middlea=440
      midiformiddlea=69
      basetone=2**((1/12.)*(-midiformiddlea))*middlea
      toptone=2**((1/12.)*(127-midiformiddlea))*middlea
      baseperiodinsamples=samplefreq/basetone         # 2398.2785461781523
      shortestperiodinsamples=samplefreq/toptone      # 1.5631434496286949
      #baseoctave=[2**24/(samplefreq/(basetone*2**(i/12.))) for i in range(12)]
      topoctave=[2**24/(samplefreq/(toptone*2**(-i/12.))) for i in range(12)]
      topoctaveint=[int(round(v)) for v in topoctave]
    So, for maximum precision, we use a table with the top frequency steps,
    such that the very top value is tone 127.
 */

#if KNUDSEN_IS_A_PAIN
#define KNUDSEN_CRIPPLED 1
uns24 step;
#endif
#if KNUDSEN_CRIPPLED
#else   // Test program
int step, Carry, W;
#endif

void midi2step(int x) {
#pragma updateBank 0
        char shift=(128-12)/12+1;       // octaves down from top to tone 0
        W=x;
        W-=(128-12)%12;                 // offset within octave to end at top note

        // Count which octave the note is in (max 10 iterations)
        // incidentally calculate the modulo
#if KNUDSEN_CRIPPLED
#else   // Test program
        Carry=W>=0;
#endif
        if (Carry==1) {         // Not the bottom few notes
                do {
                        shift--;       // count octaves
                        W-=12;
```

```
#if KNUDSEN_CRIPPLED
#else   // Test program
                        Carry=W>=0;
#endif
                } while(Carry==1);      // Carry is set while W sign stays positive
        }
        W+=12;          // get back to range 0..11

        // Look up base tone (power of two/octave multiple) to transpose down
        x=W;            // store index
#if KNUDSEN_CRIPPLED
        x+=W;           // *2   (carry now 0)
        x=rl(x);        // *4
        x=rl(x);        // *8           // Knudsen manual EXPLICITLY mentions W=rl(x)!
        W=x;            // but it doesn't compile, so we get a load here
        skip(W);        // Each row below is exactly 8 instructions
        { step= 5685608; goto got_step; nop(); }
        { step= 6023692; goto got_step; nop(); }
        { step= 6381879; goto got_step; nop(); }
        { step= 6761365; goto got_step; nop(); }
        { step= 7163417; goto got_step; nop(); }
        { step= 7589376; goto got_step; nop(); }
        { step= 8040664; goto got_step; nop(); }
        { step= 8518786; goto got_step; nop(); }
        { step= 9025340; goto got_step; nop(); }
        { step= 9562014; goto got_step; nop(); }
        { step=10130601; goto got_step; nop(); }
        { step=10732998; goto got_step; nop(); }
        // 3 W=literal, 3 store W, 1 goto, 1 nop
got_step:
#else
        //printf("shift %d, x %d\n", shift, x);
        // Knudsen does particularly poor switch code,
        // turning it into an if..else if chain.
        switch (x) {
                case  0: step= 5685608; break;
                case  1: step= 6023692; break;
                case  2: step= 6381879; break;
                case  3: step= 6761365; break;
                case  4: step= 7163417; break;
                case  5: step= 7589376; break;
                case  6: step= 8040664; break;
                case  7: step= 8518786; break;
                case  8: step= 9025340; break;
                case  9: step= 9562014; break;
                case 10: step=10130601; break;
                case 11: step=10732998; break;
        }
#endif

        // Transpose down to correct octave
```

```
        if(shift) {
                do {
#if KNUDSEN_CRIPPLED
                        step.high16>>=1;
                        step.low8=rr(step.low8);        // rotate in carry flag
#else
                        Carry=step&1;
                        step>>=1;
#endif
                } while (--shift);
                if (Carry==1) {         // Round upwards
#if KNUDSEN_CRIPPLED
                        // hack: our table never has 8 1s in a row, no need for carry
                        step.low8+=1;
#else
                        step+=1;
#endif
                }
        }
}
#pragma updateBank 1

#if KNUDSEN_CRIPPLED
// No test code for you, PIC!
#else
#include <stdio.h>

const float samplefreq=20e6/4/255, bittwentyfour=256*256*256;

void testnote(int i) {
        midi2step(i);
        printf("Midi note %d step %d freq %g Hz\n", i,
                        step, samplefreq/(bittwentyfour/step));
}

int main(int argc, char *argv[]) {
        int i;

        puts("Number 69 should be 440Hz. Others in order, lowest 8.176, highest 12.54e3.");
        testnote(0);
        testnote(7);
        testnote(8);
        testnote(9);
        for(i=0x45-2; i<0x45+12; i++) {
                testnote(i);
        }
        testnote(127-12);
        testnote(127-11);
        testnote(127);
        return 0;
}
```

```
#endif
```

This code is written to file `midi2step.c`.

8.2.1 Performance

Inspection of the generated assembly, combined with careful optimization, has brought this routine to under 100 machine cycles. In particular, Knudsen did a poor job of both const arrays and switch statements when runtime was important.

8.3 MIDI receive state machine

MIDI parsing is handled in the main loop, where an optimized state machine parses note on and off commands. Once completely received, they are handled by the `processmidi` function, while resets are handled by `resetmidi`.

17a ⟨*declarations.c* 5⟩+≡ (4a) ◃5 20a▹

```
char midistate, midinote, midibyte, playing[CHANNELS];
bit midion @ midistate.4, midiexpectvelocity @ midistate.6,
        midionoff @ midistate.7;

// Subroutine for handling MIDI input, load data in midibyte
void processmidi(void);
void resetmidi(void);
```

17b ⟨*setup-midi-parser.c* 17b⟩≡ (9)

```
resetmidi();    // reset MIDI state
```

18a ⟨*uart-receive.c* 18a⟩≡ (11)

```
if (RCIF) {
        // Handled MIDI commands: ff reset, 8x note off, 9x note on
        lightdiode=!lightdiode;         // indicate traffic
        W=RCREG;                // read MIDI byte
        midibyte=W;
        if (midibyte&0x80) {    // command byte
                midistate=0;    // default: ignore bytes until next command
                W&=0x60;
                if (W==0) {     // note on or off
                        if ((midibyte&0x0f)<TABLES)     // allowed channel
                                midistate=midibyte;     // expect note & velocity
                } else {
                        if (++midibyte==0) {    // Reset
                                resetmidi();
                        }
                }
        } else {
                if (midionoff) {
                        if (midiexpectvelocity) {
                                processmidi();
                                midiexpectvelocity=0;
                        } else {
                                midiexpectvelocity=1;
                                midinote=midibyte;
                        }
                }
        }
}
```

18b ⟨*resetmidi.c* 18b⟩≡ (12)

```
void resetmidi(void) {
        // Stop all sounds
        int channel;
        for(channel=0; channel<CHANNELS; channel++) {
                steps[channel]=0;
                phase[channel]=0;
                playing[channel].7=1;   // not playing
                bank[channel]=TABLEBASE;
        }
        CCPR1L=127;             /* "neutral" duty cycle */
        midistate=0;
}
```

19 ⟨*processmidi.c* 19⟩≡ (12)

```
void processmidi(void) {
	int channel;
	int b=midistate&0x0f;
	if (b>=TABLES)
		return;
	b+=TABLEBASE;

	for(channel=0; channel<CHANNELS; channel++) {
		if (playing[channel]==midinote && bank[channel]==b)
			break;  // found that note playing
	}
	if (midion==0 || midibyte==0) { // note off
		if (channel!=CHANNELS) {
			// Turn matching channel off
			playing[channel].7=1;   // not playing
			GIE=0;
			steps[channel]=0;
			GIE=1;
		}
	} else if (channel==CHANNELS) { // note not yet playing
		for(channel=0; channel<CHANNELS; channel++) {
			if (playing[channel].7)
				break;  // found a free channel
		}
		if (channel==CHANNELS) {
			return;         // Ignore note, already full
			// Option: replacement strategy
			//if(++lastchannel>=CHANNELS)
			//      lastchannel=0;
			//channel=lastchannel;
		}
		// start playing the note
		playing[channel]=midinote;
		midi2step(midinote);
		GIE=0;
		steps[channel]=step;
		bank[channel]=b;
		GIE=1;
	}
}
```

9 Constants

The final section of ROM contains wavetables, stored in the form of `RETLW` instructions. They are specifically placed for use with the `lookup` function, and generated by a Python script in a format Knudsen CC5X recognizes.

20a ⟨*declarations.c* 5⟩+≡ (4a) ◁17a

```
#define TABLEBASE 3     // high bits of first wavetable address
#define TABLES 4        // number of samples
```

20b ⟨*constants.c* 20b⟩≡ (4a)

```
#include "tables.c"
```

The file tables.c is generated by the Python script below.

20c ⟨*tables.py* 20c⟩≡

```
from math import *

sintable=[int(round(sin(2*pi*i/256)*127)) for i in range(256)]

tritable=[(i+128)%256-128 for i in range(256)]

squaretable=128*[127]+128*[-127]

sawtable=range(0,128,2)+range(127,-128,-2)+range(-126,1,2)

print '#include <hexcodes.h>'

# for compatibility with midi.c
print '#ifndef TABLEBASE'
print '#define TABLEBASE SINTABLE'
print '#endif'

print '/* Autogenerated wavetable data, RETLW instructions */'

def cdata(table, address):
  assert len(table)==256
  # Bias of 127 for middle of PWM range
  retlws=["__RETLW(%d)"%(v+127) for v in table]
  return "#pragma cdata[(%s)*0x100] = %s"%(address, ", ".join(retlws))

print "// Sine wave"
print cdata(sintable, "TABLEBASE")
print "// Sawtooth wave"
print cdata(sawtable, "TABLEBASE+1")
print "// Triangle wave"
print cdata(tritable, "TABLEBASE+2")
print "// Square wave"
print cdata(squaretable, "TABLEBASE+3")
```

This code is written to file `tables.py`.

10 Build system

A straightforward Makefile was constructed, which runs CC5X using Wine. As the source migrated into the report itself, Noweb is used to extract program code, and LaTeXfor documentation printouts.

21 ⟨*Makefile* 21⟩≡

```
all: synth.hex synth.asm midi2step synth.dvi

Makefile tables.py synth.c midi2step.c synth.tex: synth.nw
        noweb $<

synth.dvi: synth.tex synth.toc
        latex $<

synth.toc: synth.tex
        latex $<

%.ps: %.dvi
        dvips $<

%.hex %.asm %.cod: %.c
        # -GS?
        wine ../windows/Cc5x/CC5X.EXE -a -CC -I../windows/Cc5x $<

midi2step: midi2step.c

tone.hex tone.asm: tone.c sin.c

tables.c: tables.py
        python $< > $@

synth.hex synth.asm: synth.c tables.c midi2step.c

midi.hex midi.asm: midi.c tables.c midi2step.c

flash: synth.hex
        ( cd ../kitsrus_pic_programmer ; \
          ./picpro.py -p /dev/ttyUSB0 --pic_type=16F628A -i $(PWD)/$< )

sim: synth.hex
        gpsim -pp16f628a $<
```

This code is written to file `Makefile`.

Part III
Verification

The synthesizer can be tested in conjunction with a normal MIDI controller, such as a keyboard. Polyphonic audio can then be tested by simply holding multiple keys, and melodies should work. An LED is also present for visual verification of function.

11 Checklist

1. The LED shall not change state when no MIDI traffic is detected (keyboard off or disconnected).
2. The LED should flash with an active MIDI controller connected, at approximately 2/3Hz or higher, indicating the Active Sense messages.
3. Pressing a key on the keyboard should cause a tone to sound, and releasing it should cause it to stop.
4. The tone should be in correct pitch, i.e. 440Hz for middle A and so on.
5. Pressing multiple keys should cause multiple tones to sound simultaneously.