dsPIC Cuckoo Clock

By
Moe Wheatley
MoeTronix

www.moetronix.com

Sept. 15, 2004

Table of Contents

1 Introduction

1.1 The Idea

Since the first cave dwellers noticed the shadow of a rock moving as the day progressed, we have been developing new ways to measure and display the passing of time. http://physics.nist.gov/GenInt/Time/time.html
We now measure the oscillations of cesium atoms to keep track of our passage through the 4th dimension.

The introduction of the dsPIC seemed as good a reason as any to develop yet another clock. To make it more interesting it was decided to have the primary method to set the clock by tuning onto one of the NIST WWV or WWVH shortwave broadcasts and routing the signal to the clock to decode the time information and set itself to the correct time.

As a secondary task, the dsPIC was challenged to create various "clock" sounds such as bells, cuckoos, and gongs.

This project has its roots in an old project inspired by Dr. David Mills http://www.eecis.udel.edu/~mills/ who developed a clock using DSP techniques to extract the time information from the WWV signals.

It was also desired to evaluate the MPLAB C30 compiler and discover how well it performed so all but a couple small DSP routines were written in 'C'.

1.2 Hardware

The hardware chosen for this project was the dsPICDEM 1.1 evaluation board from Microchip. It provides a standalone platform for the clock with no hardware modifications or additions except for an external shortwave radio capable of receiving the WWV or WWVH signals on 2.5, 5.0, 10.0, 15.0 or 20.0MHz.

The SW radio audio is fed into the CODEC audio MIC input jack J16. The chimes and gong sounds are output on the CODEC audio SPKR OUT jack J17.

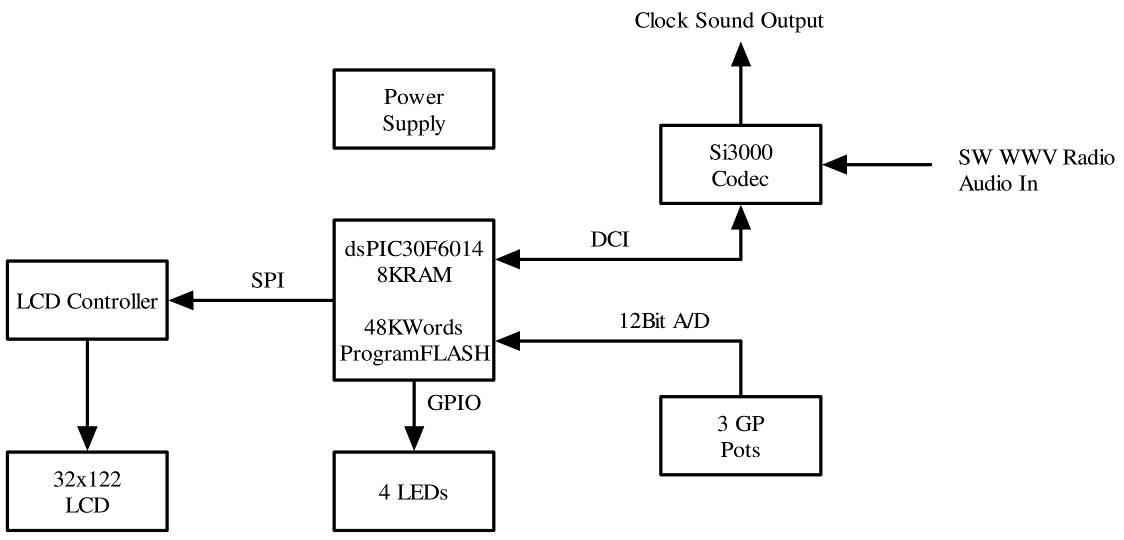
Two of the three available pots, RP1 and 3 are used to set the radio input audio level and sound out audio levels. They are hooked to the dsPIC 12bit A/D converter and read periodically and the values used to adjust the CODEC parameters.

The LCD and 4 key switches SW1-4 are used to implement a simple menu and display system.

Four LEDs, LED1-4 provide visual status of the clock synchronization and audio level clipping.

DsPIC resources used are a couple timers, the 12 bit A/D converter, the DCI module, the SPI module, internal EEROM, and some I/O ports.

1.2.1 Block Diagram



1.3 Software

The software is written primarily in C using the Microchip MPLAB C30 Compiler. Debug and program development was accomplished using the MPLAB IDE and the ICD-2 programming/debug module along with an oscilloscope.

1.3.1 Basic Modules

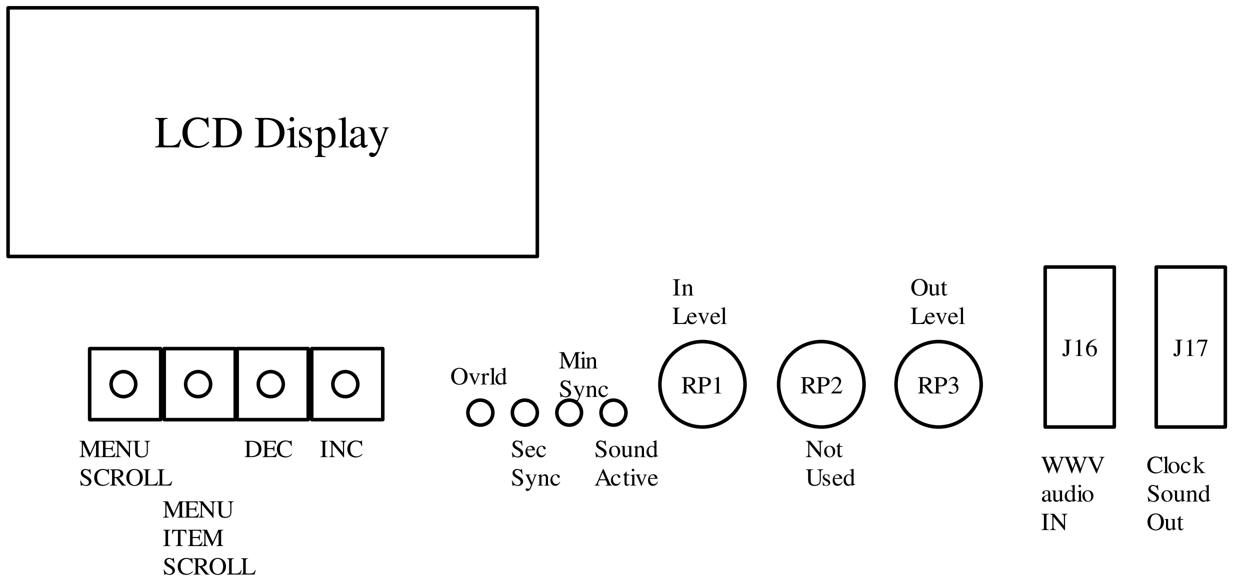
There are nine software modules that are linked together. The following is a list of the modules and their basic functions:

Main.c	Program Entry and main execution loop
Timers.c	Timer setup, IRQ service, and EEROM utilities
AtoD.c	12 bit A/D setup and utilities
Codec.c	Si3000 Codec setup and IRQ service routine
LCD.c	LCD controller setup and interface routines
Menu.c	Menu system implementation
Gettime.c	Decodes time information from raw bit data
ProcSignal.c	DSP section which demodulates the WWV or WWVH signals
SoundGen.c	Creates sound samples for all the various clock sounds
AsmUtils.s	A couple of assembler code helper functions for the DSP tasks

The following include files are also used:

Timers.h	Module Includes
AtoD.h	Module Includes
Codec.h	Module Includes
LCD.h	Module Includes
Menu.h	Module Includes
Gettime.h	Module Includes
ProcSignal.h	Module Includes
SoundGen.h	Module Includes
Common.h	common program definitions
Tables.h	DSP filter tables
SoundTable.h	FM synthesis sound tables
Cuckoo.*	Various project files for MPLAB

2 User Operation Guide



2.1 Program Loading

First the MPLAB IDE, the MPLAB C30 Compiler, and ICD-2 software must be installed and the ICD-2 attached to the PC and to the dsPICDEM 1.1 evaluation board as described in the documentation for the tool sets.

Create a folder and place all the Cuckoo project files into it.

From within the MPLAB IDE, load the project file Cuckoo.mcp into the IDE workspace. If the C30 Compiler is setup then the project can be compiled and the demo board programmed with the compiled code in either debug mode or just use the programming mode.

2.2 Radio Connection

An AM shortwave receiver is required to pick up the WWV or WWVH signals. Hopefully it will have an external headphone or speaker jack that can be used to connect to the Demo Board's Mic input jack.
Tune the receiver to 2.5, 5.0, 10.0, 15.0, or 20.0 MHz whichever has the best signal. An outdoor antenna is preferred but a wire strung around the room may suffice if the signal is reasonably strong.

An alternative is to use an MP3 file that is available from www.moetronix.com/files/wwv.mp3

This is a 5 minute recording of WWV that can be played back on your computer's soundcard into the demo board in order to demonstrate the program.

2.3 Initial Setup

Once the program is loaded and running, a splash screen should appear followed by the main menu screen displaying the time and a pendulum moving at the bottom of the screen.

If headphones or a speaker is connected to the SPKR jack of the demo board, some initial clock gongs will be heard. Adjust RP3 for the output volume.

With the radio connected and receiving WWV, adjust RP1 (and/or the radio volume) until LED1 just flickers indicating overload, then back it off till it just stops blinking. (A better way is to punch SW1 three times till the raw signal view is shown and adjust for a signal that is not touching the top or bottom of the screen)

2.4 Menus

There are seven menus that can be reached by pressing the Menu Scroll button(SW1)

2.4.1 Main

The main menu displays the time as 24 hour UTC and also as 12 hour local time. A cursor pendulum bounces back and forth at the bottom of the screen.

2.4.2 Setup

The setup menu allows user setup data to be entered.

```
LOCAL HR OFFSET= -04     (-12 to +12 hours)
STATION =  WWV or WWVH
HOUR CHIME = CUCKOO or GONG
TIC-TOCK = ON or OFF
```

A blinking cursor appears near the parameter that can be changed. Pressing the Menu Item Scroll Key(SW2) moves the cursor to select each menu item that can be changed.

Pressing the DEC or INC keys (SW3,SW4) changes the selected menu item.

All the setup items as well as the current selected menu is saved in non-volatile EEROM onboard the dsPIC.

Note: the hourly gongs will chime if the Local Hr Offset is changed even though it is not at the top of the hour. This is a quick and dirty way to force an hourly chime to adjust the volume or just to play with the chimes.

2.4.3 Status

The status menu displays some inner program status for following the synchronization and data integration processes.

```
SYNC Y or N    MIN QUAL 00
MIN ENERGY  00
HR ENERGY   00
YR ENERGY   00
```

The Sync parameter indicates that the seconds position is located and synchronized.
The Minute Quality indicates how many minute sync pulses have been received correctly.
The Minute, Hour, and Year Energies indicate the number of correct bits that have been integrated over the last several minutes. The larger the energy, the more likely the time and data are correct. Above a certain threshold or if sync is lost, the energies are reset and a new capture is started.

2.4.4 Manual Time Set

This menu allows the time to be set manually in case the WWV signal is not available. The year, UTC hour and minute can be changed. Whenever the minute value is changed, the seconds count is forced to zero so one can manually sync to within a few seconds.

```
      MANUAL TIME SET
Year = 04
UTC Hour = 21
Minute = 59
```

A blinking cursor appears near the parameter that can be changed. Pressing the Menu Item Scroll Key(SW2) moves the cursor to select each menu item that can be changed.

Pressing the DEC or INC keys (SW3,SW4) changes the selected menu item.

These time values are also saved in non-volatile EEROM onboard the dsPIC. Only the values last changed in this menu are saved, not the current time.

2.4.5 Raw Signal View

The raw signal view displays the incoming audio signal on an oscilloscope type display. This is useful for adjusting the radio signal level so that it does not clip or that it is strong enough.

2.4.6 Seconds Sync View

The seconds sync view displays the bin energies of the seconds sync buffer. This buffer integrates the 1 second "Tick" pulse energy in a 150 position buffer where each bin is a 6.6666mS time slot of a full second. The peak value of this buffer represents the seconds sync position and is displayed at the center of the screen. It is a measure of the quality of the incoming WWV signal. If the peak is small or moving around, the signal is not sufficient to decode.

2.4.7 Data View

The Data View screen displays the current data pulse graphically. The screen width is one second. A zero data bit is a short bit taking up the first 1/5 of the screen. A one bit is about 3/4 of the screen long. This view also indicates the signal quality. If a bit cannot be discerned visually, chances are the program will not decode it either.

3 WWV Signal Description

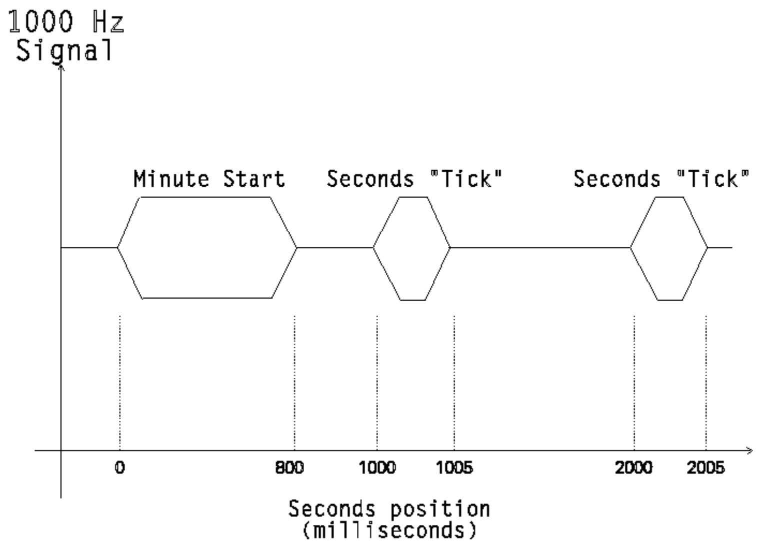
The best source for WWV signal format is to visit their web page. There is also a lot of historical as well as information on their stations and other time services.

http://www.boulder.nist.gov/timefreq/index.html

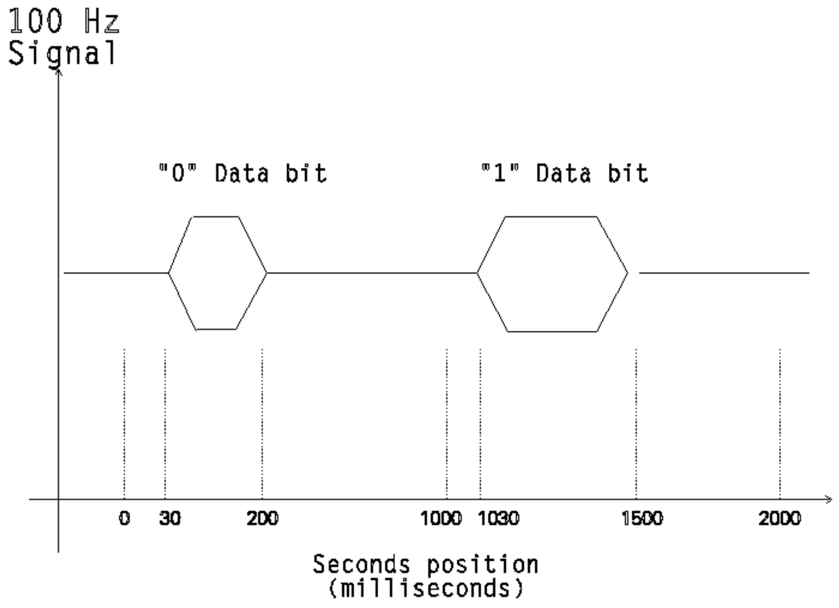
The WWV signal uses double sideband AM where four types of information is modulated onto the carrier. Voice information is modulated at 75%, the steady tones are at 50%, the second's "tick" sound is 100%, and the BCD time data is 25%. For this project only the 1000/1200Hz seconds/minute identifier tones and the 100Hz BCD time data are utilized.

Time data is sent using pulses of 100Hz tones at a data rate of 1 Hz in BCD format.

Various tone sequences are broadcast by WWV. The start of each hour is identified with a .8 second burst of 1500Hz tone. The start of each minute is identified with a .8 second burst of 1000Hz tone. The start of each second is identified with 5 cycles of 1000Hz tone. (Except for the 29th and 59th second) WWVH uses 1200Hz tones instead of 1000Hz tones, otherwise the timing is all the same.

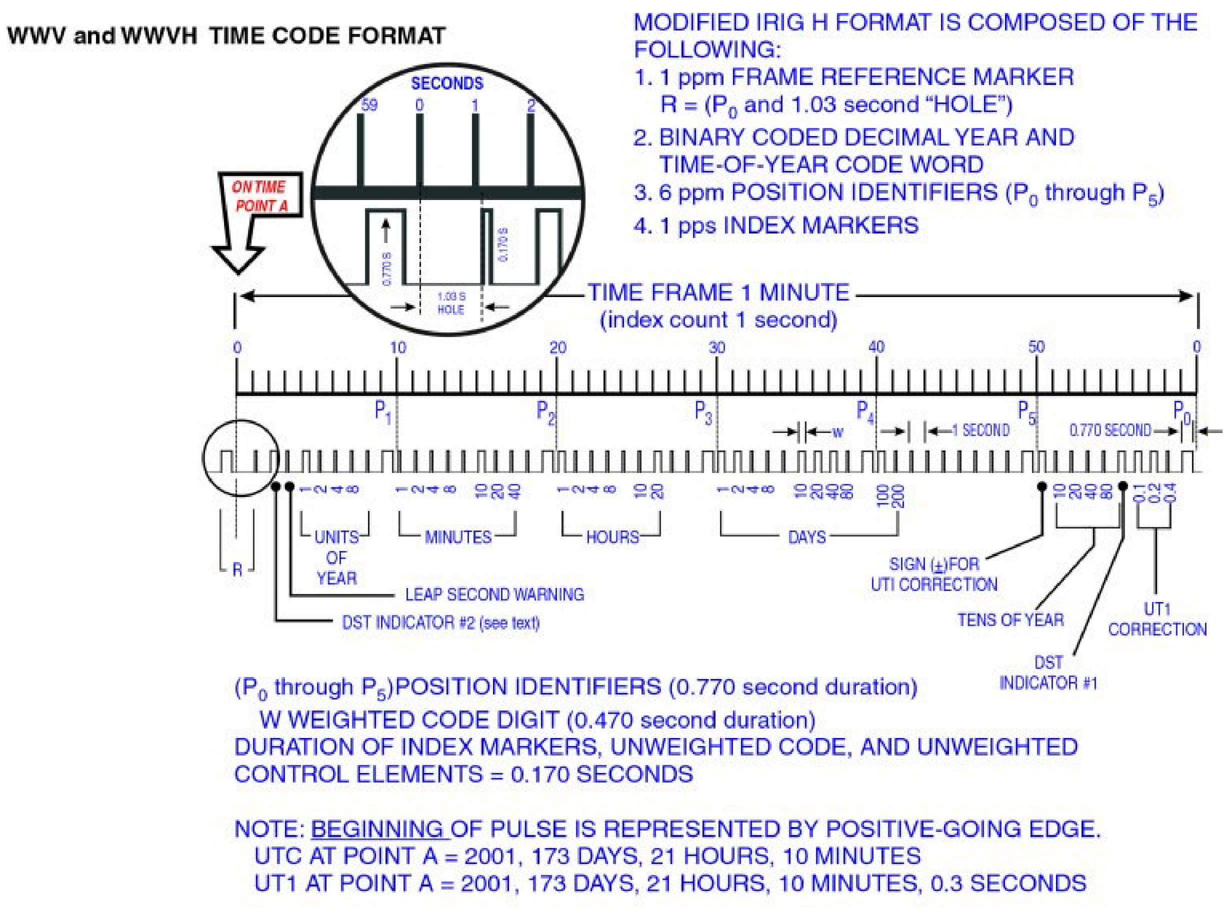


Time data is sent using pulses of 100Hz tones at a data rate of 1 Hz in BCD format.



The following shows the data bit definitions and locations within each minute:

WWV and WWVH TIME CODE FORMAT



MODIFIED IRIG H FORMAT IS COMPOSED OF THE FOLLOWING:

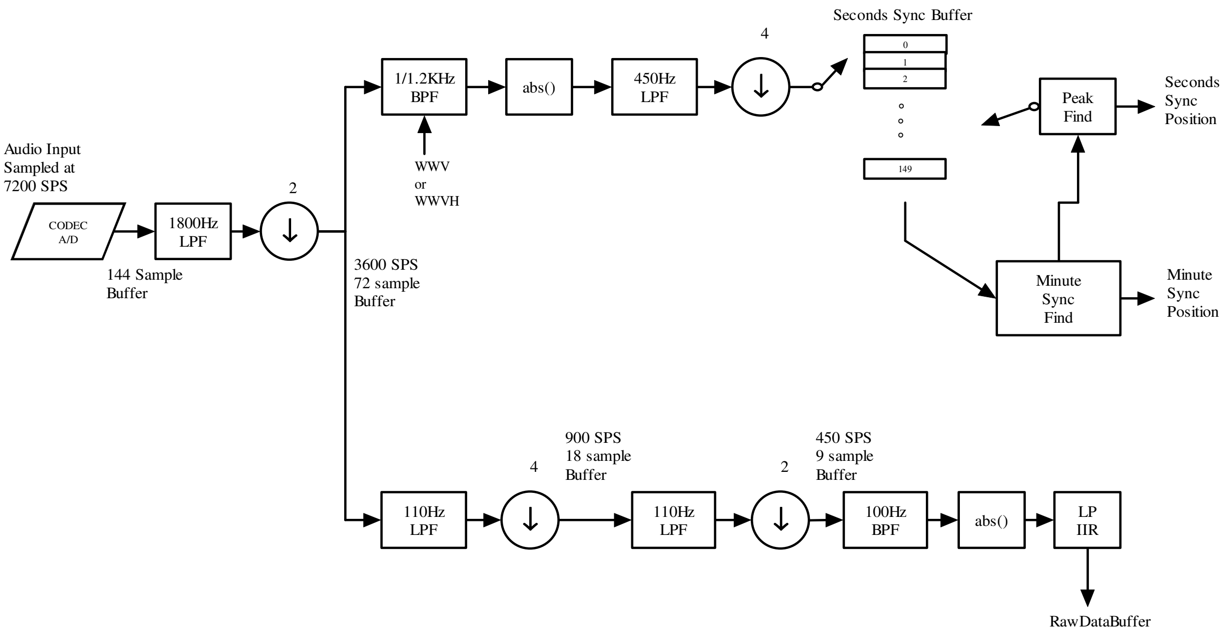
1. 1 ppm FRAME REFERENCE MARKER R = (P_0 and 1.03 second “HOLE”)
2. BINARY CODED DECIMAL YEAR AND TIME-OF-YEAR CODE WORD
3. 6 ppm POSITION IDENTIFIERS (P_0 through P_5)
4. 1 pps INDEX MARKERS

(P_0 through P_5) POSITION IDENTIFIERS (0.770 second duration)
W WEIGHTED CODE DIGIT (0.470 second duration)
DURATION OF INDEX MARKERS, UNWEIGHTED CODE, AND UNWEIGHTED CONTROL ELEMENTS = 0.170 SECONDS

NOTE: BEGINNING OF PULSE IS REPRESENTED BY POSITIVE-GOING EDGE.
UTC AT POINT A = 2001, 173 DAYS, 21 HOURS, 10 MINUTES
UT1 AT POINT A = 2001, 173 DAYS, 21 HOURS, 10 MINUTES, 0.3 SECONDS

4 WWV/WWVH Demodulator

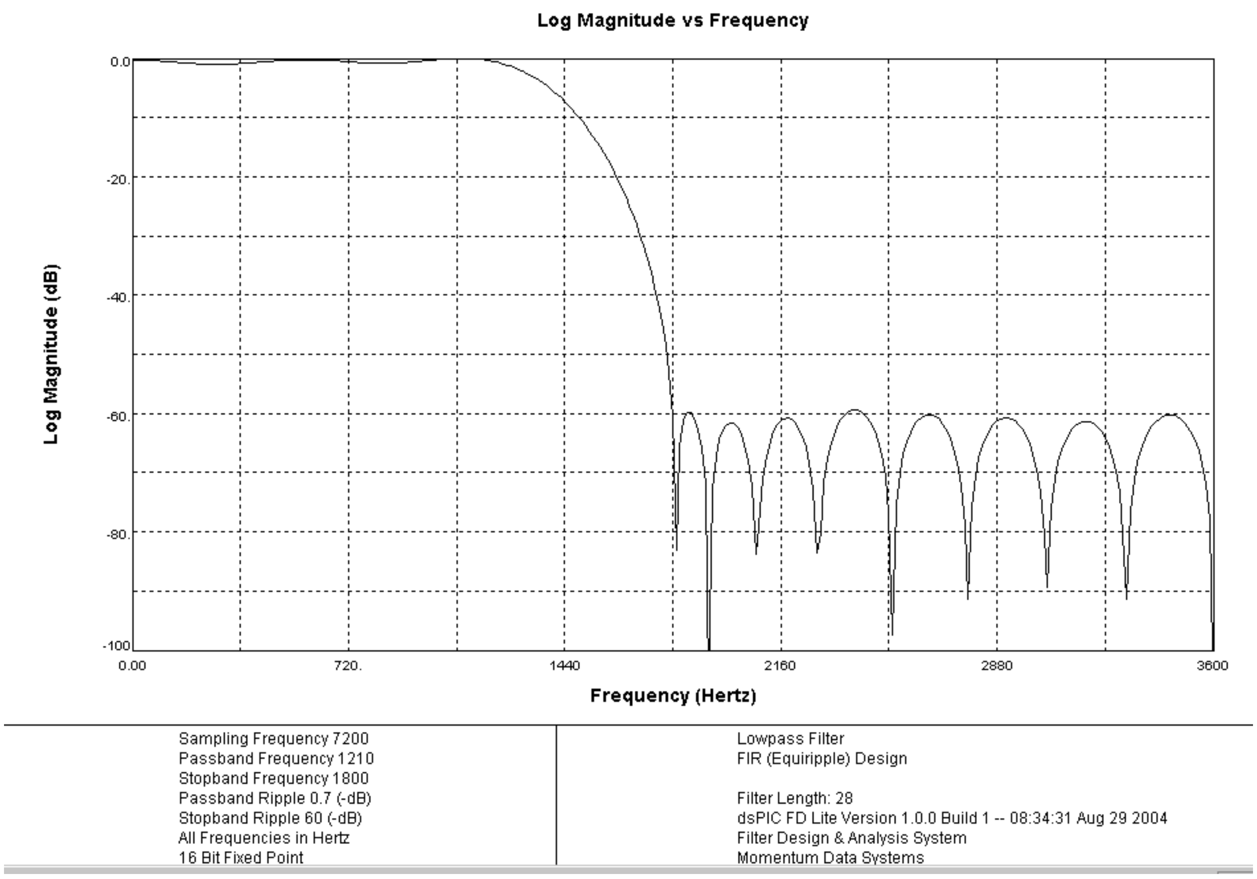
WWV/WWVH Signal Processing Block Diagram



4.1 First Decimation Stage

The Si3000 CODEC provides 15 bit samples of input audio data at a rate of 7200 SPS. The codec.c module contains the service routines for the codec. A ping-pong buffer scheme allows one buffer to be processed while the other one is being filled. Also a state machine is implemented to allow the codec parameters to be changed on the fly without disturbing the incoming and outgoing data samples. One must change the bits per frame of the DCI to insert the extra setup frame in between the data sample frames.

Since the highest frequency component of the WWV signal is <1500Hz, the data stream is first decimated by two. This is accomplished using the DSP library decimate routine and supplying it with anti-aliasing LP filter coefficients that provide a 60dB cutoff at 1800Hz. Below is a dsPIC Filter Designer screen shot of the filter magnitude.



Sampling Frequency 7200	Lowpass Filter
Passband Frequency 1210	FIR (Equiripple) Design
Stopband Frequency 1800	
Passband Ripple 0.7 (-dB)	Filter Length: 28
Stopband Ripple 60 (-dB)	dsPIC FD Lite Version 1.0.0 Build 1 -- 08:34:31 Aug 29 2004
All Frequencies in Hertz	Filter Design & Analysis System
16 Bit Fixed Point	Momentum Data Systems

The decimated data stream splits into two paths, one to demodulate the 1000 (or 1200,WWVH) Hz sync tones and one to demodulate the 100Hz time data bit tones.

4.2 Data Decimation and Detection Stages

The data path consists of two additional decimation stages to reduce the sample rate down to 450Hz. A decimate by 4 followed by a decimate by 2 stage accomplishes this task. This 450 SPS data stream is then band pass filtered at 100Hz to isolate the 100Hz data signal. It is then AM detected using an absolute value function then low pass filtered again using a simple IIR filter.

The low pass filter is a simple IIR stage with one delay element. It has the same response as an analog RC filter. This type filter is useful for obtaining a very low cutoff frequency with little processor overhead. It can be implemented with one line of code.

```
y = k1*y + k2*x;
```

The filter H(z) equation is:

$$Hs(z) := \frac{k2 \cdot z}{z - k1}$$

The down side of this filter is that it has poor frequency roll off response but is fine for implementing long running averages.

The Cuckoo clock implements this filter as an array of IIR filters with a small assembly language utility routine in AsmUtils.s called

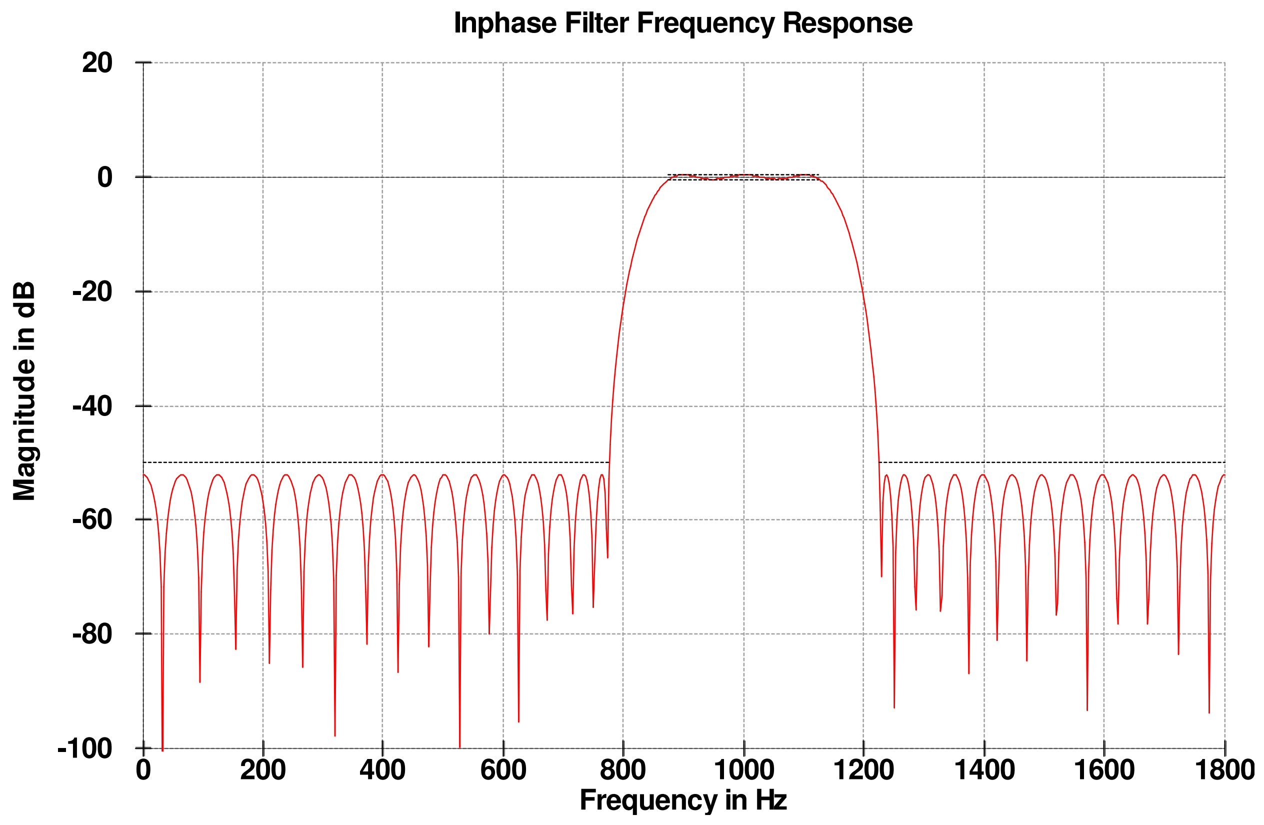
void CalcLpIir(unsigned int Size, fractional* pIn, fractional* pLPVal, unsigned int LPK).
LPK is the filter coefficient where:

k1 = 1.0 - LPK and k2 = LPK

This probably could be done in 'C' but it was desired to put the MAC and MPY instructions to good use.

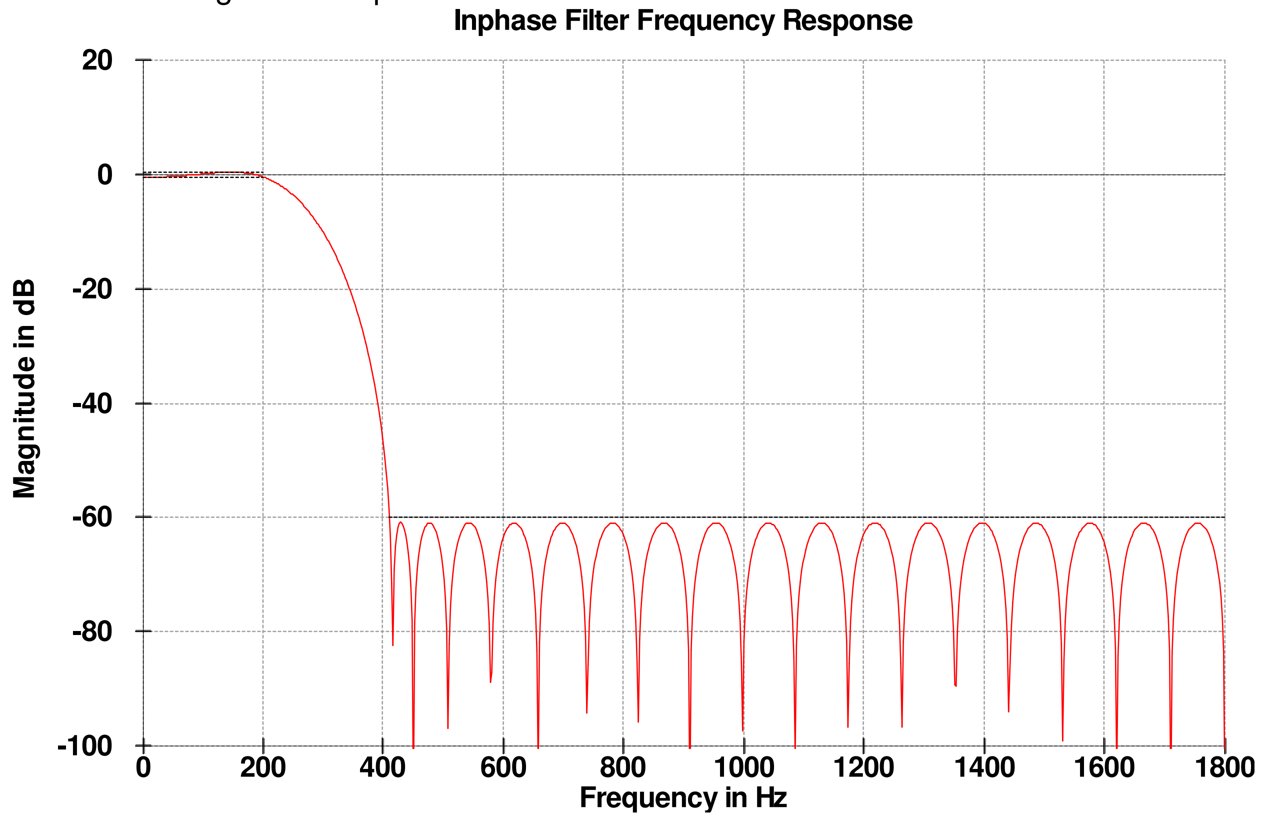
4.3 Seconds Sync Recovery

The 1000Hz (1200Hz for WWVH) signal is obtained by band-pass filtering the 3600SPS data stream with an 81 tap BP FIR filter. Since the pulse width of the seconds sync pulse is 5mSec, the filter band width needs to be roughly 1/5mS or 200Hz. Below is the magnitude response of the filter.



A separate filter for WWVH is used that is similar except is centered at 1200Hz.

This filtered signal is then AM detected using an abs() function and then decimated by a factor of 4 to a final sample rate of 450SPS. The decimation LP filter has a cutoff of 200Hz and magnitude response is shown below.



The seconds sync detection scheme works by implementing a 150 position energy collection buffer where each location represents the averaged energy in 6.666.. mSec steps. The seconds sync data is added to each "time bin" so that a full seconds worth is captured in the entire buffer. Each bin's data is LP filtered using the IIR filter described earlier. After several seconds of averaging, if a seconds sync pulse is present, one or two bins will contain a lot more energy than the bins being filled with just noise. The IIR LP filters allow averaging over many seconds and greatly increase the signal to noise ratio of the sync pulses.

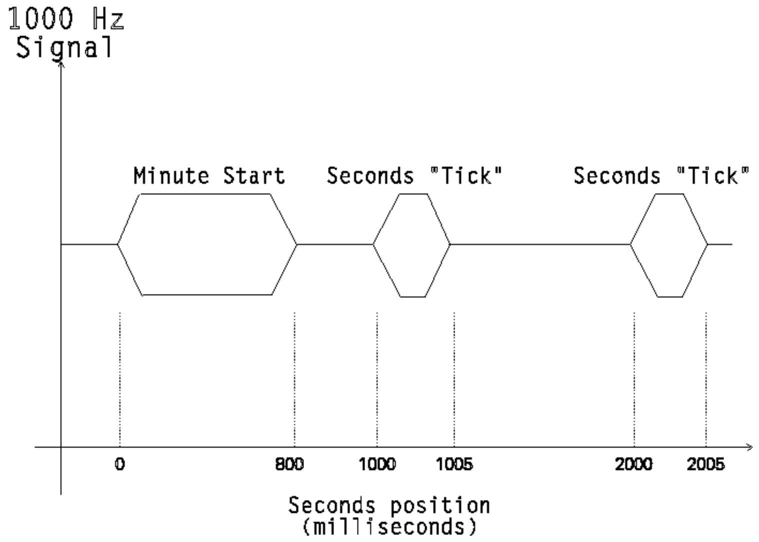
By simply calling the DSP library VectorMax(..) function, both the value of the peak energy as well as the peak location within the 150 position SecSyncBuf can be obtained. This position can now be used as the reference position in time for data recovery and also Minute sync recovery.

One of the LCD Menu screens displays this buffer so that one can visually see the sync pulse activity. It is normalized so that the peak is always shown at the center of the screen.

Since it is possible(probably) that a sync pulse will straddle two time bins, a means to slightly adjust the data clock was implemented to keep the sync pulse centered within one bin. This routine simply looks on either side of the peak bin and either adds a dummy sample into the Codec data stream or skips one sample. Eventually the sync pulse will center itself within one bin.

4.4 Minute Sync Recovery

Once the seconds sync position is known, the minute sync position can be determined by comparing the 1000Hz energy in the first 800mSecs of each second interval with the energy in the last 200mSec. If a minute sync tone is present, the integrated energy over the 800mSec will be much larger than that of the last 200mSec where there is no tone. This ratio provides a S/N ratio that when processed and qualified determines the start of each minute.

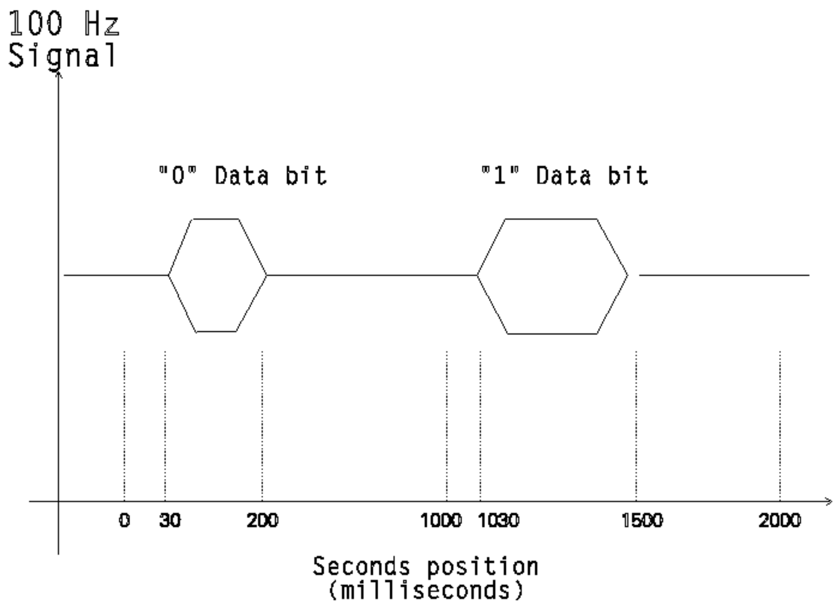


Once found, the sync routine keeps track of additional minute sync pulses and increments a sync quality variable. If a sync pulse is found in a different seconds location than the original, the quality variable is decremented. If the value reaches zero it is assumed the minute position is wrong and it starts over.

4.5 Data bit Recovery

Once the seconds and minute sync positions are determined, actual time data can start being collected. A zero bit consists of 100Hz energy at 30mSec from the second start to 200Msec from the start. A one consists of energy from 30mSec from the seconds start to 500mSec from the start. By adding the energy from the 100Hz tone over the zero interval and comparing it to the energy over the last 200ms noise interval, a judgment can be made as to a zero or no signal. Likewise by integrating over the 300 to 500mSec range, a one bit can be determined.

The routine "CalcBit()" returns a -1, or a +1, or a 0 for a zero bit, a one bit, and a no bit found respectively.



5 Time Decoder

The time data is decoded in the gettime.c software module. This module manages the actual time decoding as well as incrementing and calculating local time variables.

The seconds counter counts from 0 to 59 and is used as a pointer to the position of each specific data bit that needs to be found. Each seconds location corresponds to a specific time data bit. A simple jump table was implemented that calls the appropriate routine to process the desired bits. If a bit is not needed, a NOP() function is called.

In a perfect world it would only take one minute to capture all the data bits and be able to set the time of the clock. However the WWV signal is seldom perfect and a more robust scheme is needed to obtain the data bits correctly. The method used is to integrate each bit over several minutes until they reach a threshold where the probability that they are correct is good. Since a zero is -1 and a one is +1 and an unknown bit is a 0, one only has to add up the bits in each seconds location and the sign of the sum will be the final bit value. This would work great except for one problem. Time does not stand still and as every minute passes, the minute and possibly all the rest of the time data will change so you can't just add up the bit positions any more.

The solution is to add all possible bit combinations of each new data bit to an array and then search for the maximum value within the array which, over time, should be the best guess for the correct time.

Since there are 38 data bits in each minute, one would need an array of 2^38 locations to store all possibilities. Since 275 Gbytes is a little steep, only a subset of all the data bits is used. If only the time and year bits are used, 24 bits are required but this is still way over the memory budget. Since the hour and year doesn't change very often one can just look at the minutes, hour, and year bits as small groups and use much smaller arrays.

Since the data is BCD encoded, there are actually only 60 possible combinations for minutes, only 24 combinations for hours, and 100 combinations for the last 2 digits of the year. By splitting up the integration arrays in this manner, it easily fits within the dsPICs memory space.

The UpdatxxxBit() routines are called to place the latest bit value into these integration arrays. Routines to check for the peak integration value are called on second positions where there are no data bits to collect. Thresholds are checked and when reached, the time is set. During the integration process, the display shows its best guess as a way to see how well the integration process is proceeding. On strong signals, the correct time is usually found within a couple minutes even though the thresholds may not be reached for many more minutes.

6 FM Synthesizer

Just displaying the correct time would be boring so some clock sounds are generated on the 15 minute marks as well as on the hour. A tick tock sound is also available on every second.
If there was sufficient Flash memory available, it would be a trivial task to just read out pre-recorded audio samples. One way would be to add a little eight pin 8Mbit Flash chip to the dsPICDEM 1.1 board. Since the goal was to use an unmodified board, a different method would be needed.

In the 1970's a new method of synthesizing sound was created by John Chowning at Stanford University. His research an techniques led to the development of the very popular Yamaha DX-7 music synthesizer.

http://www.harmony-central.com/Computer/synth-history.html

A search on the web will reveal much more detail on the FM synthesis techniques so only a brief description will be given here.
Basically the idea is to FM modulate a sine wave with another sine wave where the amplitude of both can be modified in time. Sounds with complex harmonic content can be easily generated using this technique.

The basic generator is

$V(t) = A(t)\sin(2\pi Fct + MI(t)\sin(2\pi Fmt))$

The outer sine is the main carrier tone while the inner sine is the FM modulating tone. The four main parameters are the carrier frequency(Fc), the carrier amplitude(A(t), the modulation frequency(Fm), and the Modulation Index(MI(t).
 Note the carrier and modulation frequency are fixed in time but the amplitude and modulation index are time varying.
Multiple generators like this can then be added together for very complex sound structures.

SoundGen.c and the associated soundtable.h file implement all the cuckoo clock sounds. The sound generator when triggered by time events, generates 7200SPS 16 bit audio samples that are then output to the Si3000 Codec D/A.

The routine ServiceSound() is called to fill a new Codec output buffer and monitor the time events for triggering the various sounds and SoundStateMachine() is called to initialize the required sounds and repeat the hour chimes. The function CreateNextSample() is the guts of the generator and creates a single sample that is the sum of three of the FM generators.

Each generator uses the following data structure for its parameters:

```
typedef struct
{
        int mphzinc;          //modulation phz increment value
        int mphzacc;          //modulation phz accumulator
        int mindexslope;      //modulation index slope value
        int mindex;           //modulation index
        int cphzinc;          //carrier phz increment value(.10986 Hz increments)
        int cphzacc;          //carrier phz accumulator
        int campslope;        //carrier amplitude slope value
        int camp;             //carrier amplitude
} FMGenStruct;
```

The sin generator is implemented using a 16 bit accumulator which has the phase increment value added to it each sample time. The top 8 bits of the accumulator are then used as an index into a 256 position sine lookup table.
Carrier and modulation frequency is then set by setting the phase increment value where the frequency is F*65536/7200.

In order to reduce the size of the sound parameter tables, the slope of the amplitude and modulation index are stored in the tables. This makes the table generation more awkward but makes the tables much more compact.

Each sound requires two data structures. The first is an initialization table that provides the starting values for the three generators:

```
typedef struct
{
      int minc0;             //Gen 0 initial modulation phz increment value
      int mindx0;            //Gen 0 initial modulation index value
      int cinc0;             //Gen 0 initial carrier phz increment value
      int camp0;             //Gen 0 initial carrier amplitude value
      int minc1;             //Gen 1 initial modulation phz increment value
      int mindx1;              //Gen 1 initial modulation index value
      int cinc1;               //Gen 1 initial carrier phz increment value
      int camp1;               //Gen 1 initial carrier amplitude value
      int minc2;               //Gen 2 initial modulation phz increment value
      int mindx2;              //Gen 2 initial modulation index value
      int cinc2;               //Gen 2 initial carrier phz increment value
      int camp2;               //Gen 2 initial carrier amplitude value
}FMInitStruct;
```

The second structure provides the run time shape of the sound and is of the following structure:

```
typedef struct
{
      int changepos;    //When the change should occur(1/7200 Sec increments)
      int mslope0;            //FM gen 0 modulation index slope value
      int caslope0;           //FM gen 0 carrier amplitude slope value
      int mslope1;            //FM gen 1 modulation index slope value
      int caslope1;           //FM gen 1 carrier amplitude slope value
      int mslope2;            //FM gen 2 modulation index slope value
      int caslope2;           //FM gen 2 carrier amplitude slope value
} FMDataStruct;
```

Basically the first parameter is the position in time where these parameters should be in effect in sample counts. The rest of the values are used until the change position is reached then it moves to the next set. When a zero is found in the changepos value, the sequence is finished.

The Cuckoo sound is the easiest to generate and doesn't really involve the FM process. It is just a single tone with an amplitude ramp up and then down followed by a second sequence at a different frequency.

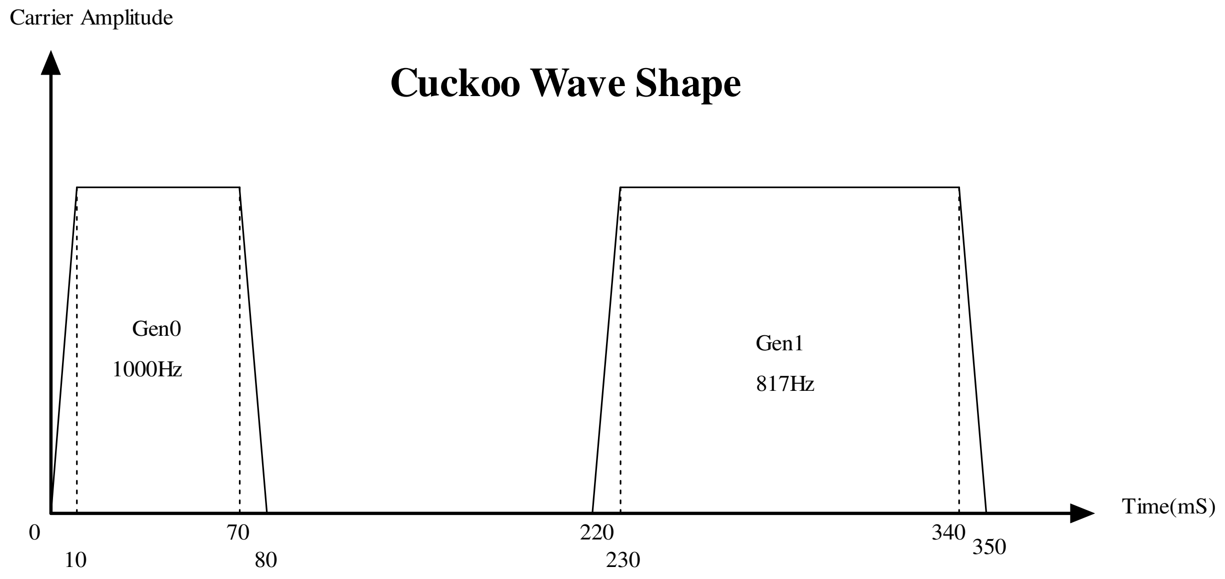
The following is the initial setup for the cuckoo sound. The amplitude starts at zero and the two desired tone frequencies are set.

```
// Cuckoo Sound  1000Hz and 817Hz
const FMInitStruct CUCKOO_INI =
{//minc, mindx, cinc,    camp
     0,   0,   9102,     0,   //gen 0
     0,   0,   7439,     0,   //gen 1
     0,   0,   0,        0    //gen 2
};
```

This is the sound shape table for the cuckoo sound.

```
const FMDataStruct CUCKOO_DATA[8] =
{//  pos, mslope0, caslope0, mslope1, caslope1, mslope2,
caslope2
       72,  0,   444,  0,      0,  0,     0,   //10mS ramp up
      504,  0,     0,  0,      0,  0,     0,   //60ms hold
      576,  0,  -444,  0,      0,  0,     0,   //10ms ramp down
     1584,  0,     0,  0,      0,  0,     0,   //140ms gap
     1656,  0,     0,  0,    444,  0,     0,   //10ms ramp up
     2448,  0,     0,  0,      0,  0,     0,   //110ms hold
     2520,  0,     0,  0,   -444,  0,     0,   //10ms ramp down
     0,0,0,0,0,0,0
};
```

The waveform looks like the following:

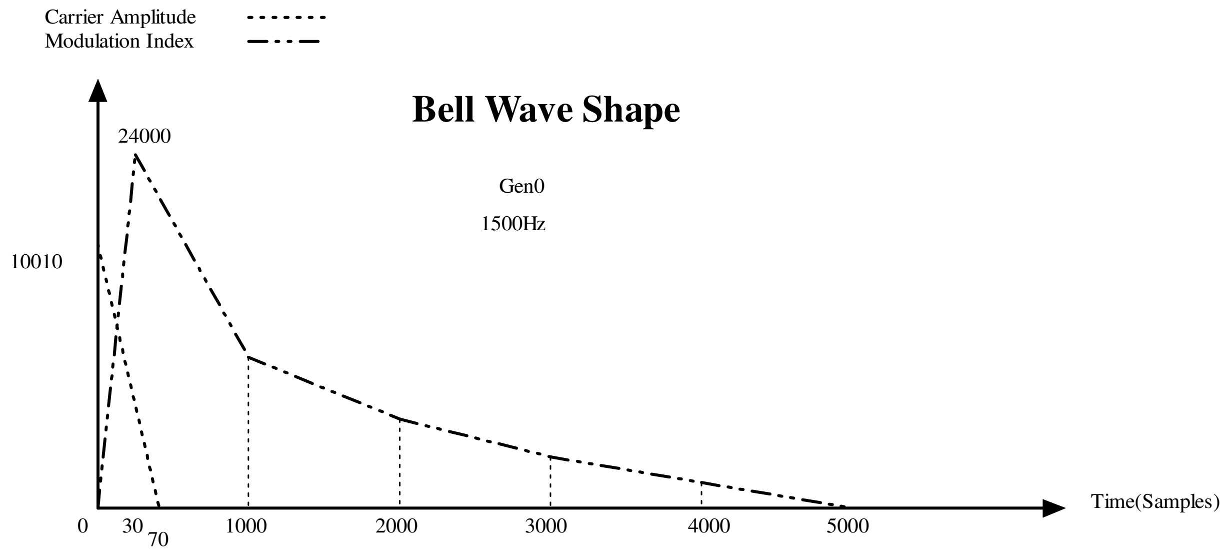


The single bell sound uses one generator with some FM modulation to create the initial harmonics that occur when a bell is struck. After that the single tone amplitude shape is a linear piece-wise approximation to a decaying exponential.

```
//================================================
// 1 Bell Sound  1500Hz
const FMInitStruct BELL1_INI =
{//minc, mindx, cinc, camp
     9751,10010,13653,   0,   //gen 0
     0,   0,   0,        0,   //gen 1
     0,   0,   0,        0    //gen 2
};

const FMDataStruct BELL1_DATA[8] =
{//pos, mslope0, caslope0, mslope1, caslope1, mslope2, caslope2
     30,  -143,     800,      0,0,0,0,  //4mS
     70,  -143,     -12,      0,0,0,0,  //10ms
     1000,  0,      -12,      0,0,0,0,  //
     2000,  0,      -6,       0,0,0,0,
     3000,  0,      -3,       0,0,0,0,
     4000,  0,      -2,       0,0,0,0,
     5000,  0,      -1,       0,0,0,0,
     0,0,0,0,0,0,0
};
```

The carrier and modulation index waveforms for the bell sound are graphed below:



The Gong sound is the most complex and requires all three generators at different frequencies along with different modulation index waveforms.

7 Summary

This project served its purpose in gaining a better understanding of the dsPIC and its development tool set.

7.1 Processor Resources

The clock is completely functional and did not consume all the dsPIC resources in terms of CPU power or memory.
The program used about 17Kbytes of program memory and a little over 3Kbytes of data RAM.

The worst case CPU load in the main non-interrupt processing loop was about 32% at the slowest dsPICDEM 1.1 board clock rate of 7.3728MHz and an x4 PLL.
The interrupt background tasks add about another 5% to the CPU load.

One of the biggest surprises on this project was the implementation of the sound generators. It was assumed that the FM generators would have to be implemented in assembler in order to take advantage of the Q1.15 fractional multiplier and accumulator as well as the repeat instruction. The algorithm was coded in C just to verify the approach before hand translating to assembler. It turned out that the C compiler generated code was more than sufficient to implement the generators without having to revert to assembler. Looking at the assembler generated code reveals pretty decent optimizing of the 32bit casts and multiplication code.

7.2 Clock Performance Issues

7.2.1 WWV/WWVH Demodulation

The WWV demodulator/decoder is far from optimum. The ideal demodulator would involve matched filters for extracting the sync and data pulses. Unfortunately, there is not enough on-board data RAM to implement the long correlation arrays needed. However, since at least in North America, WWV can be heard most of the time with decent signal integrity so the clock can be set fairly quickly.

The demodulator algorithm is susceptible to noise and interference on the WWV signal. False minute sync is a problem during periods of high static or even loud voice announcements. A separate narrower minute sync filter would probably help this area.

7.2.2 Time Keeping

The time keeping when not synced to WWV, depends upon the accuracy of the main CPU clock oscillator. One possible solution would be to monitor the clock accuracy while synced up to WWV and store this value in non-volatile EEROM and use it to add or subtract samples to maintain better clock accuracy.

There is a processing delay so the clock will be behind the true time by a hundred mSecs or so. This could be factored out in order to make the clock tics match the WWV ticks exactly.

7.2.3 Sound Generation

The sound generation could be improved with a lot more tinkering. Adding more steps to the sound tables would smooth out the decay envelopes. Also more accurate sounds could be created with lots of tweaking of the FM parameters. Adding an external Flash memory chip to save actual waveforms would be a better product solution.

7.3 Toolset Issues

The MPLAB IDE and C30 compiler operated with few problems. The ICD-2 debugger worked as long as one didn't forget to plug it into the demo board before programming. Usually a program restart and power sequence of the target board and ICD-2 would be required of that occurred.

Most of the debugging was done with a scope and not the IDE. Single stepping was problematic especially when interrupts were flailing away.