"TIC" Technical Description

Project Description

The TIC software project was an experiment in using DSP methods to demodulate WWV time signals and synthesize various clock sounds using the TAPR/AMSAT DSP-93 platform.

Some of the TIC implementation features:

- AGC function on input samples.
- 1000Hz and 100Hz FIR Bandpass filters used to recover WWV time signals.
- Bit by bit data integration method extracts time information from the WWV signal.
- ASCII time string output in 12 or 24 hour format from DSP-93 UART.
- Hourly gongs or cuckoo's and bells at 15, 30, and 45 minutes past hour.
- Sound effects produced using FM synthesis methods.
- Automatic Daylight savings time adjustment except for GMT output.
- Assembly options for various time zones, sound options, radio port, etc.
- Automatic internal clock compensation for stand alone timekeeping.

Radio Station WWV is a shortwave radio station operated by the National Institute of Standards and Technology and broadcasts time data as well as various voice announcements on 2.5, 5, 10, 15, and 20 MHz. out of Ft. Collins, Colorado. For more details of this station you can check out the web site,
http://www.boulder.nist.gov/timefreq/pubs/sp432/s_wwv.htm>

Another station WWVH is located in Kauai Hawaii but uses slightly different tones. Since it is not received clearly here on the east coast, this project was developed to only decode WWV.

The WWV Signal

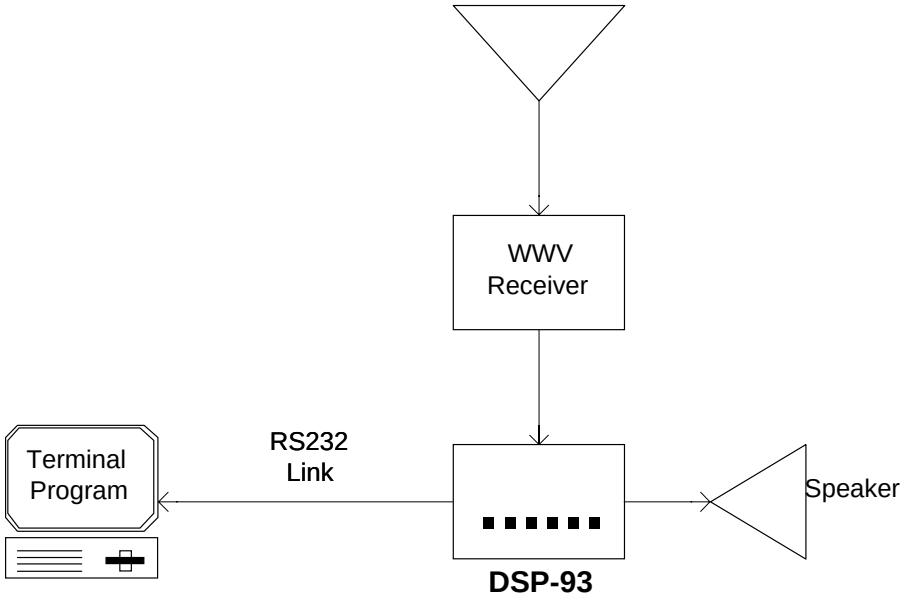
Various tone sequences are broadcast by WWV. The start of each hour is identified with a .8 second burst of 1500Hz tone. The start of each minute is identified with a .8 second burst of 1000Hz tone. The start of each second is identified with 5 cycles of 1000Hz tone. (Except for the 29th and 59th second) Time data is sent using pulses of 100Hz tones at a data rate of 1 Hz in BCD format.

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 1000Hz seconds/minute identifier tones and the 100Hz BCD time data are utilized.

Hardware Platform

The DSP-93 platform consists of a 40 MHz Texas Instruments TMS320C25 16 bit DSP chip, surrounded by 32K words of program memory and 32K words of data RAM. An analog interface board contains a TI TLC32044 14 bit A/D, D/A converter, a asynchronous UART chip, and various I/O ports for radio control and LED display control. A software controllable gain block is provided for adjusting the receiver input level to the A/D converter. A monitor EPROM is used to provide a downloader function as well as storing built in modem software and test applications. Programs can be downloaded into the DSP-93 using utility programs that run on a PC.

The following diagram shows the basic setup for using the DSP-93 to receive WWV signals. The DSP-93 connects to the radio using one of the radio ports. A little more information is given in the file TIC1USR.PDF concerning setup of the cuckoo clock.



Software Design Method

The software for the TIC1 clock was designed in a modular fashion using “C” language blocks to describe each function. Once the code blocks were defined, then the C320-25 assembly code was hand assembled using the “C” code blocks as a reference. This may seem cumbersome but designing in assembly language can get very complicated and confusing in a hurry even with generous comments. By designing the code in a higher level language, one doesn’t get bogged down in implementation details until the design is ironed out. This may take a little longer to get to the debug stage, but reduces the number of bugs once you get there, especially as the program gets more complicated.

The software source was also broken into several parts mainly to ease in editing. This sort of implements a “poor man’s” linking assembler in that one can edit and debug

using just the file associated with a general task instead of having to search through one large cumbersome source file. When assembling of course, all the files have to be re-assembled.
Data queues(FIFO, Circular, or “rubber band” buffer) are used on the A/D and D/A channel to reduce the timing constrains on the software and allow even distribution of processing time.

The only time critical operation is the actual A/D and D/A sampling operation which is performed by the TLC32044 CODEC chip in conjunction with the DSP hardware. Since the data is taken from or put into the data sample queues at precise time intervals, the rest of the software is not constrained to operate on the data in real time. This allows the software to perform periodic operations longer than the sample time interval as long as the average processing time does not exceed the sample time interval.

Another software method used was the use of indirect function calls to implement state machines which are of use at various places in the code. Basically the address of the function to call is placed in a RAM variable. An indirect call using that RAM variable results in execution of the specified function. Within that function, a “NEXT STATE” can be specified by simply loading the RAM variable with the address of the next state function. In this manner, complicated state machines can be implemented fairly easily.
The software is broken into six files for easier manipulation:

- TICMAIN.ASM This is the main entry file and is the one that is specified for assembling as it has all the include references for the auxiliary files. It contains constant definitions, variable allocations, hardware and software initialization, interrupt service routines, and low level bit twiddling functions. The main code service loop also resides here which calls all the other modules in a round robin fashion to service all the various tasks of the modem.

- TICDATA.TBL This file contains various constant data tables used throughout the program. A SIN table, lookup tables, FIR coefficients, sound tables, state tables, etc. are contained here.

- TICINPUT.ASM This file contains all the routines that service the A/D input samples as they arrive and demodulate it.

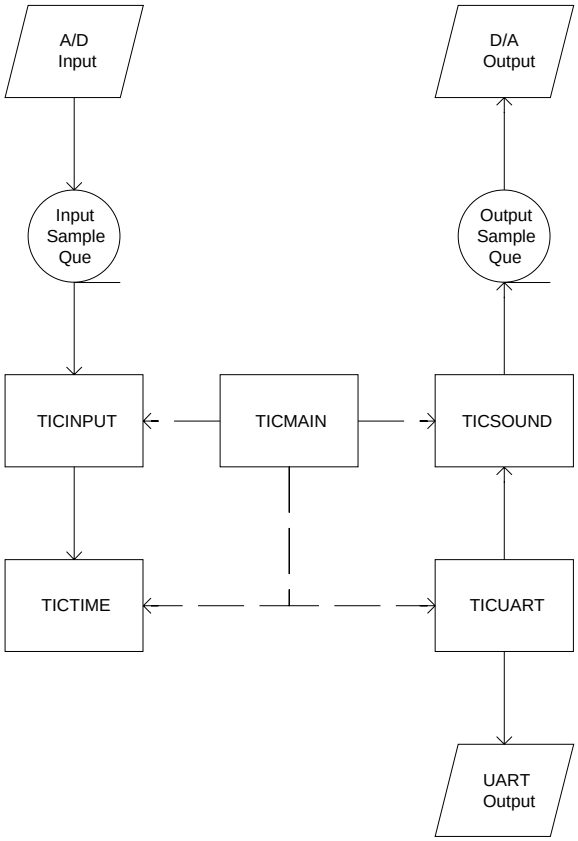
- TICTIME.ASM This file contains routines that decode time information from the WWV signal.

- TICSOUND.ASM This file contains routines that create the sounds used in the cuckoo clock.

- TICUART.ASM This file contains routines that send status and time ASCII strings out the UART port.

Software Descriptions

The software begins executing after being downloaded by first initializing the hardware resources on the DSP-93. The TLC32044 AIO chip is initialized to run at a sample rate of 10000 sample per second. The 16C550 UART chip is initialized to 19200 bps. This is the default serial output rate. The onboard timer of the TMS320C25 chip is set to interrupt every 5 milliseconds. This is used as a timer for some of the initialization routines, and synthesized time generation. Several initialization routines are called to initialize various variables used by each module. After initialization, the interrupts are enabled, and the main service loop is entered in which all four software modules are called in a loop continuously.

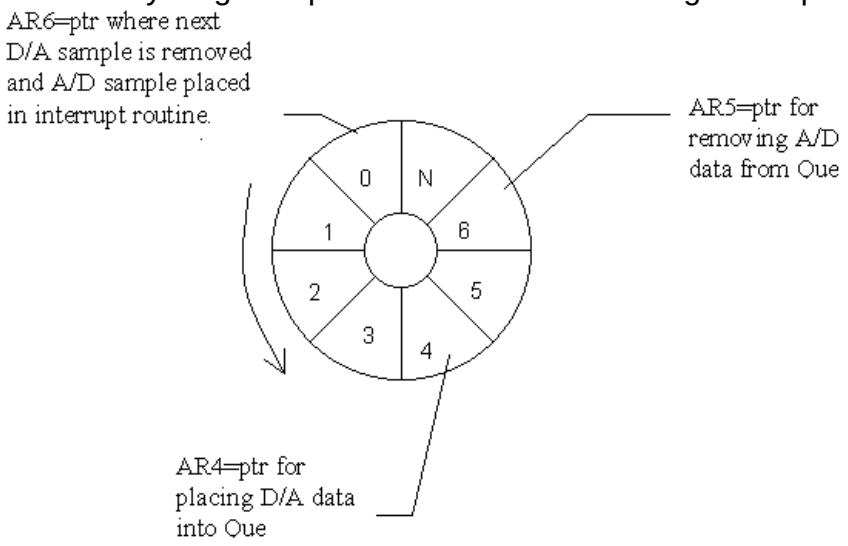


First the AIO interrupt service routine will be described. It's function is to send a new sample from the Sample_Que out the D/A PORT and store a new A/D sample into the Sample_Que. Since the A/D and D/A are run at the same sample rate, only one interrupt service routine is used for both. Also since one word is removed and one word is placed in the Sample_Que at every sample time, only 3 pointers are need to maintain the Sample_Que.

```
void RxIntService(){
   Save_Context();
   DXR = Sample_Que[AR6];              // write D/A from Sample_Que
   Sample_Que[AR6++] = DRR;            // read A/D into Sample_Que
```

```
   if( AR6>Sample_Que+SAMPQUESIZE-1 )  // deal with wrap around
      AR6 = Sample_Que;
   Restore_Context();
}
```

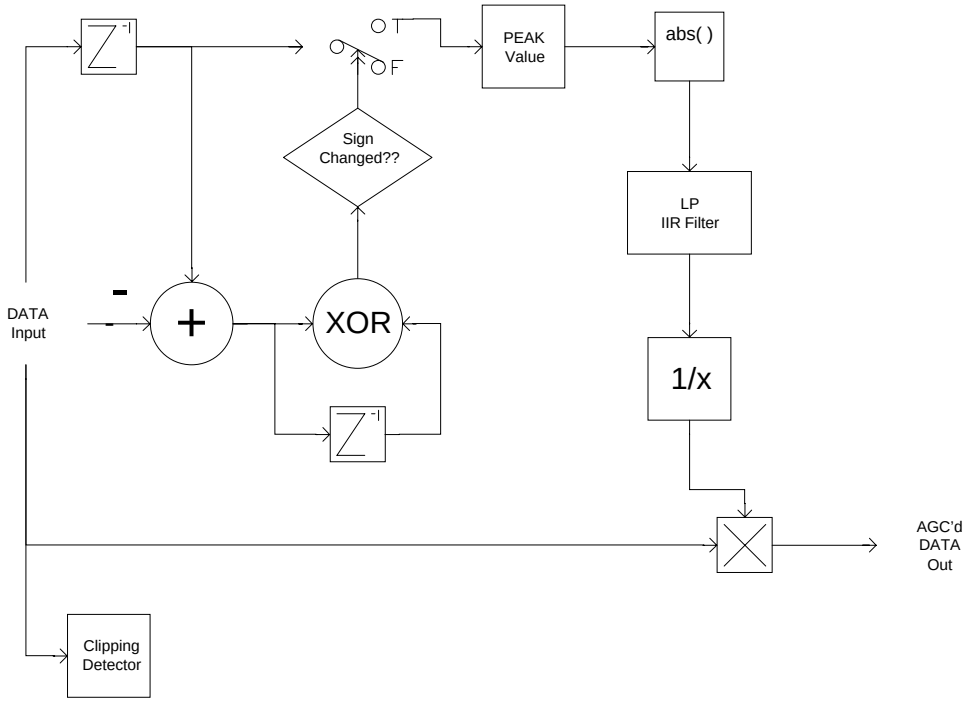
Three Auxiliary registers(AR6,AR5,AR4) are used as pointers to the Sample_Que. AR3 is used as a software stack pointer to save and restore processor context since the 320C25 doesn't save anything except the return address during interrupts.



A/D data can be removed from the Sample_Que as long as AR5 != AR6.
D/A data can be placed in the Sample_Que as long as AR4 != AR5.

TICINPUT.ASM Module

First the samples are processed by the AGC block to try and keep the amplitude constant.

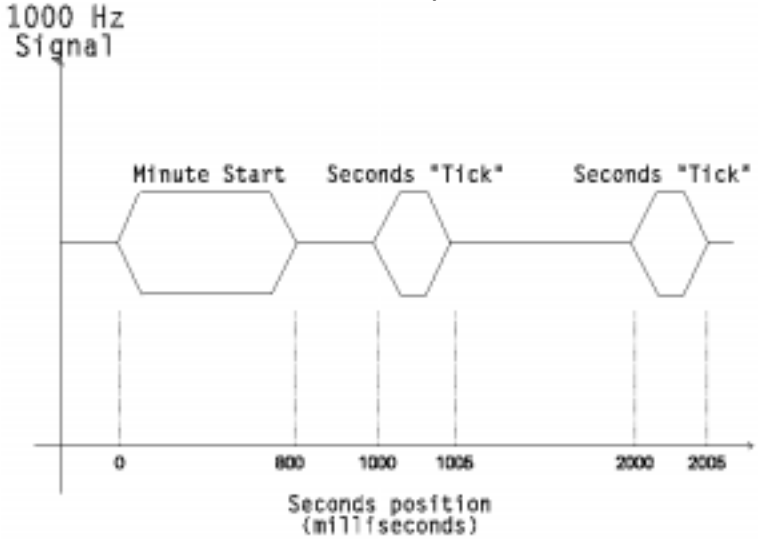


A/D samples are pulled out of the Sample_Que and passed through a clipping detector. This block just sees if any samples are above some peak threshold and flashes front panel LED7. This is useful in setting the maximum receive audio level.

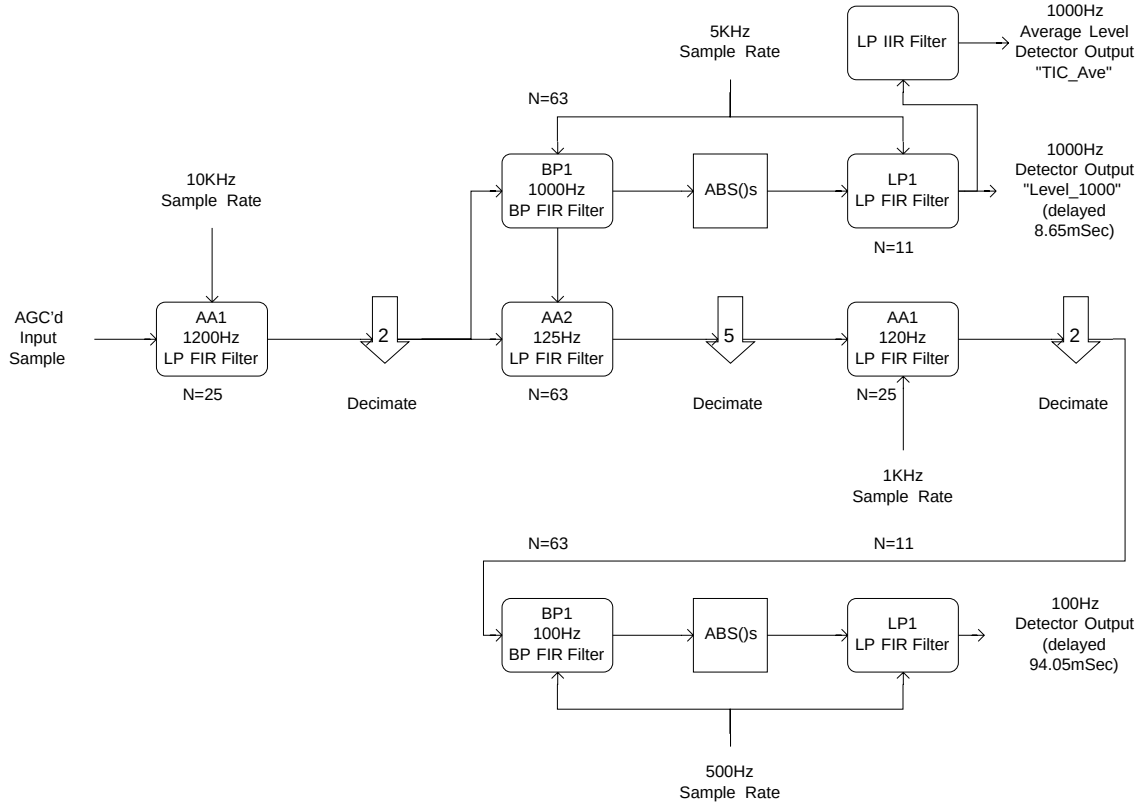
The AGC works by monitoring the incoming sample stream and calculating the slope of the incoming signal by subtracting the present sample from the previous sample. If the sign of this difference(slope) changes, then a peak in the incoming wave form has occurred. The absolute value of the sample is then stored as the signal peak level and low pass filtered. The input is then multiplied by the inverse of the low passed peak signal to obtain an AGC'd signal for the remaining signal processing.

LED6 is turned off if the input signal is below the level where the AGC can operate.

The Following figure shows the timing of the WWV 1000Hz tone. An 800 mSec burst signifies the beginning of a minute. A 5 mSec burst signifies the beginning of each second. (The 29th and 59th seconds are omitted as well as double bursts which can occur to signify UT1 time correction information.)



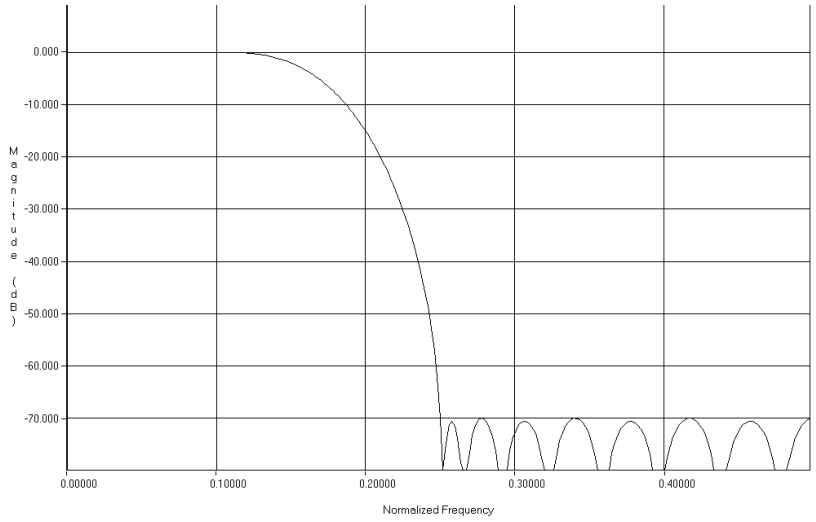
The next step is to extract the 1000Hz and 100 Hz AM signals. This is done as shown by the following diagram.

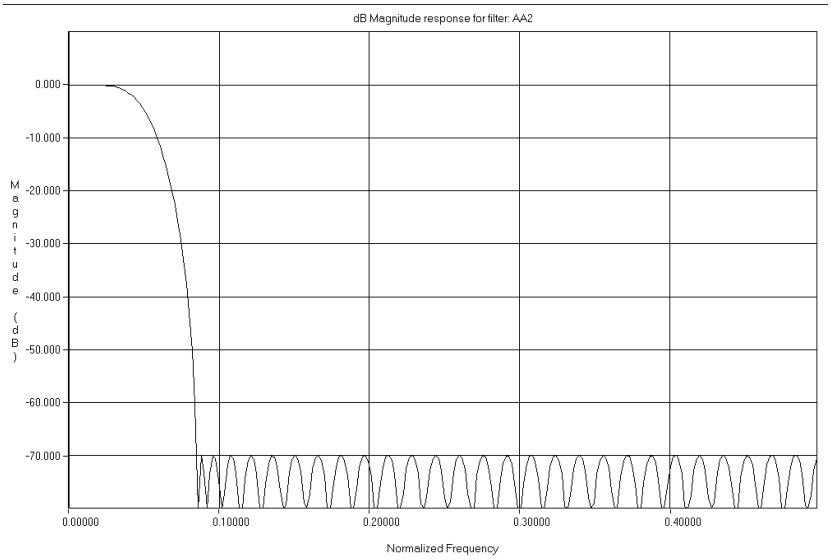


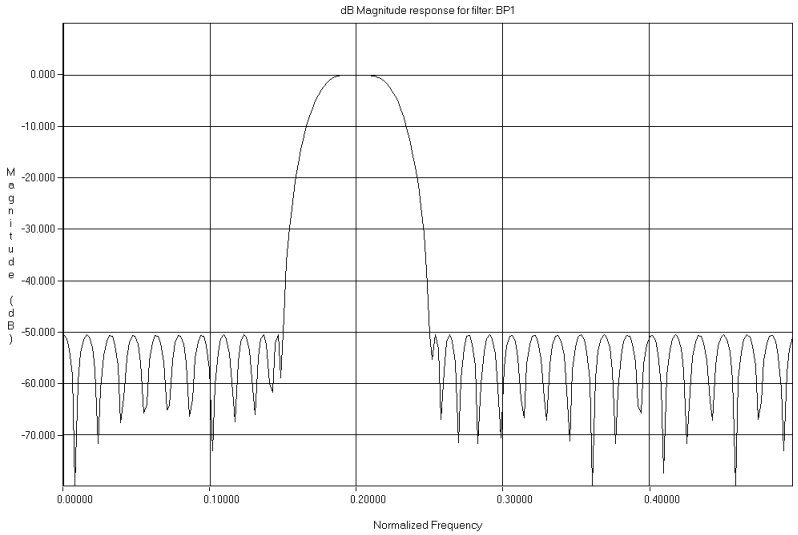
The signal's sample rate is reduced down to 5000Hz and then band pass filtered to extract the 1000Hz signal. It's then AM detected with an absolute value function then low pass filtered to extract the final demodulated 1000Hz signal.

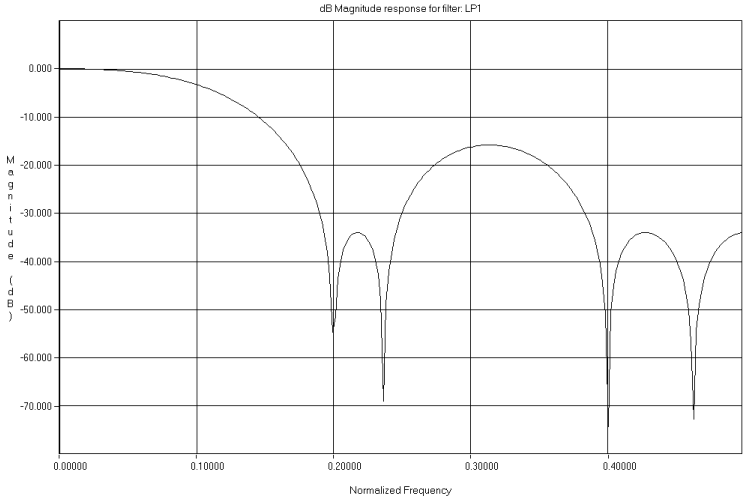
The 100Hz signal is demodulated in the same manner except that it's sample rate is decimated down to 500Hz before being band pass filtered.
The strange choice of filtering is used to be able to share delay lines and coefficients with other filter blocks. The TMS320C25 is limited in its internal memory for storing lots of filter tables. The FIR filters were designed using a program called PC-DSP from DSP Solutions. The Parks-McClellan algorithm was used in choosing the filters. The filter response curves for the filters are as follows:

AA1 FIR LP filter









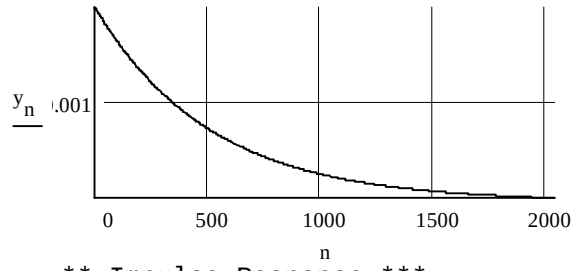
The final low pass filters are actually more like notch filters to remove the fundamental and second harmonic created from the absolute value function.

The 1000Hz output signal is further low pass filtered to obtain a long term average used as a background noise level indicator. A one pole IIR filter is used to basically keep a long term average over 512 samples or about 100mSeconds. The following is a Mathcad output of the IIR filter impulse and step response.

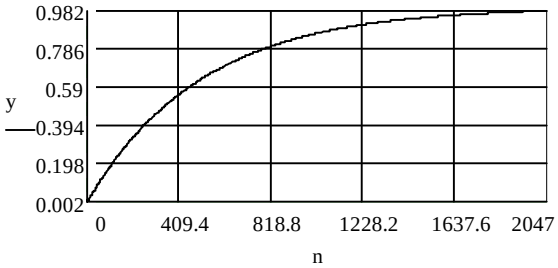
IIR Low Pass Filter

$N := 2048 \qquad n := 0 .. N - 1 \qquad ORIGIN := -1 \qquad x_n := 0 \quad y_n := 0 \quad x_0 := 1 \quad K := 512$

$$y_n := \frac{(K - 1)\cdot\left(y_{n-1}\right)}{K} + \frac{1}{K}\cdot x_n$$



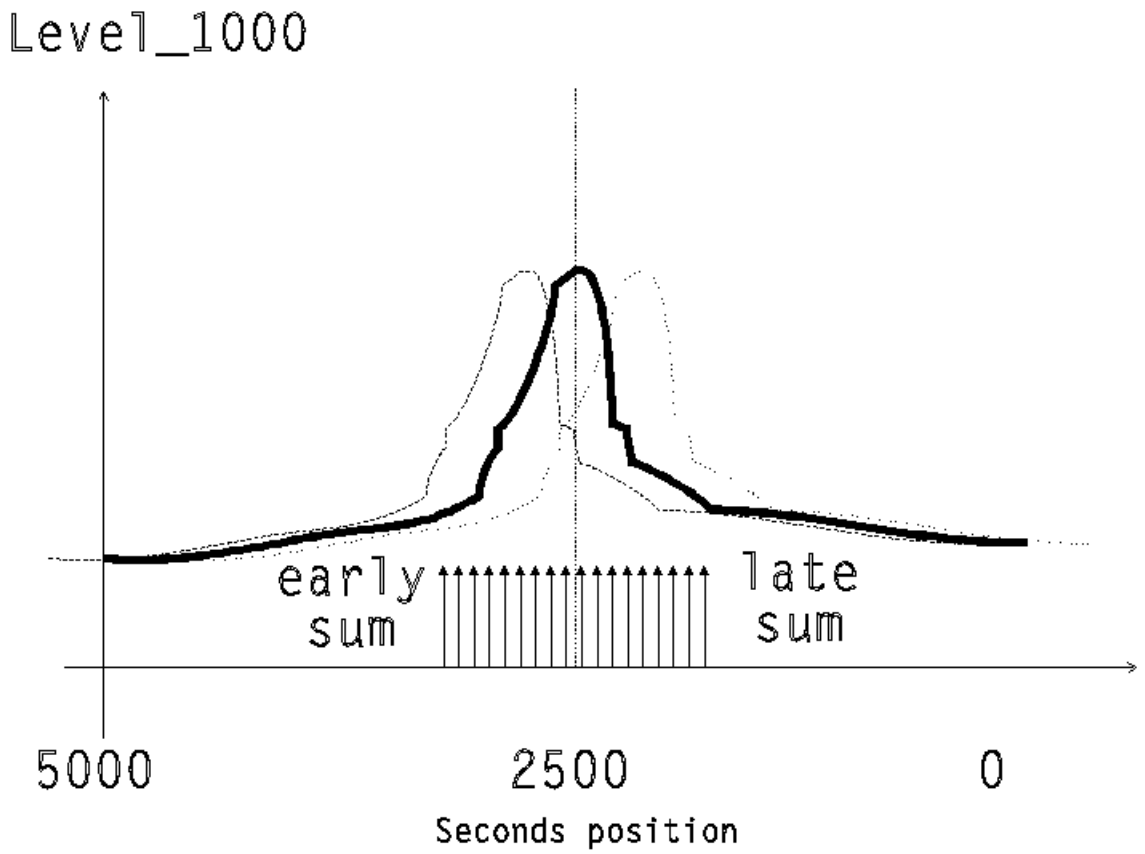
** Impulse Response ***



*** Unit Step Response ***

The last function this module performs is to find and synchronize to the 1000Hz seconds “tick” signal from WWV. If the WWV signal is just being acquired, a routine “Search_1000” is dispatched to initially find the seconds time position. This is done by creating a one second timer that counts from 5000 to zero at a 5000Hz rate. The 1000Hz signal “Level_1000” is compared to the average signal level “TIC_ave” and if it is 4 times the average, a one second one-shot timer is started and the one second timer is set to 2570. At the end of this one second interval, the signal is again compared to the average signal level and if the signal is again 4 times the backgound average, a variable “TIC_quality” is incremented. This process continues until the “TIC_quality” exceeds a threshold and then another routine “Lock_1000” is dispatched to maintain phase lock with the WWV one second tic signal.

This Lock_1000 routine samples the “Level_1000” signal for 64 early samples before and 64 late samples after the 2500 count of the seconds timer. The software then adjusts the timer value so that the sum of the 64 early samples is equal to the 64 late samples. The result is that the seconds tic pulse is centered in time around the 2500 time counter point. The total energy under the late and early regions is also compared with the background level to determine if a valid signal is present.



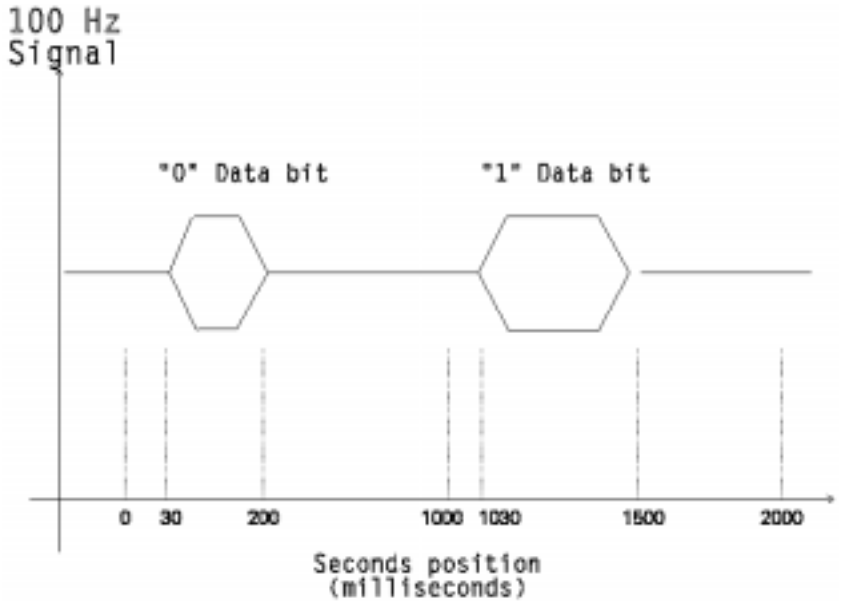
A secondary operation falls out of this method by keeping track of how many times the software has to correct for an early pulse and a late pulse. This information can be used to determine how much and what direction to adjust the DSP-93 internal oscillator. The DSP-93 oscillator is a 100ppm unit and by adjusting the internal clock, one can increase this accuracy by an order of magnitude or better. This allows the cuckoo clock to maintain accurate time after the WWV signal has been disconnected.

TICTIME.ASM Module

This is the primary module for extracting time data from the WWV signal stream. It operates at a 200Hz sample rate (5 mSec). A timer “Sec_pos” provides positional status within each 1 second interval. It is adjusted by the Lock_1000 routine to accurately time all the sampling activities.

Before any data sampling can begin, it is necessary to determine the beginning of the minute. This is done by the routine “Service_1000Hz” which finds the 800 mSec burst of 1000Hz tone and sets the seconds counter “Sec_count” to zero. This is then incremented every second up to 59 to keep track of the proper second within each minute.

WWV time data is sent during each minute interval at a 1bps rate using the following timing scheme within each 1 second interval.



The routine “Service_100Hz” samples the output of the 100Hz filter “Level_100” and by taking 200 samples over a one second interval integrates the noise and signal values to come up with a bit value based on the duration and signal to noise ratio of the 100Hz signal. The bit value takes on three possible values, +1, -1, and zero indicating a “1”, “0”, and invalid data, respectively.

The data bit definitions within each minute is shown in the following list. Numeric data is sent as BCD data. Only the minutes, hour, and Daylight savings time indicator are decoded for the cuckoo clock.

Second	Data Bit	Second	Data Bit	Second	Data Bit
0		20	Hrs 1	40	Days 100
1	Leap second	21	Hrs 2	41	Days 200
2	DST2	22	Hrs 4	42	
3		23	Hrs 8	43	
4	Year 1	24		44	
5	Year 2	25	Hrs 10	45	
6	Year 4	26	Hrs 20	46	
7	Year 8	27		47	
8		28		48	
9	P1	29	P3	49	P5
10	Min 1	30	Days 1	50	DUT+-
11	Min 2	31	Days 2	51	Year 10
12	Min 4	32	Days 4	52	Year 20
13	Min 8	33	Days 8	53	Year 40
14		34		54	Year 80
15	Min 10	35	Days 10	55	DST1
16	Min 20	36	Days 20	56	0.1 DUT
17	Min 40	37	Days 40	57	0.2 DUT
18		38	Days 80	58	0.4 DUT
19	P2	39	P4	59	P0

The basic data gathering scheme is to integrate the bit values for each data field until the total bit energy exceeds some threshold and then declare it a one or zero. By integrating each data bit field over several minutes, eventually all the data bits will be determined.

The catch to this method is that every minute the data changes as the time information advances and so several bits may reverse polarity messing up the integration sum values. The brute force method would be to create an array of all possible time values and integrate each data bit into all the possible fields and select the possible time value with the largest total integrated bit energy. For this clock which ignores day and year data, it would take 24x60x2 = 2880 time values times 14 data bits or 40,320 words to hold each bit integration value.

This is a bit much so a compromise solution was to only look at all ten possible minute unit's digits and then once the unit's digit is found, go and get the 10's digit and then the hours digits. This method takes longer but saves on memory and processing time.

A jump table is used to call the various routines depending on the particular second of the minute is current. The table is indexed by the Sec_count (0-59) and a call to each routine listed in the table is made.

```
DATASERVICE:                    ;seconds data bit service vectors
        .word   Nop             ;0
        .word   Nop             ;1
        .word   Nop             ;2   DST2
        .word   Nop             ;3   Leap sec warning
        .word   Nop             ;4   Year 1
        .word   Nop             ;5   Year 2
        .word   Nop             ;6   Year 4
        .word   Nop             ;7   Year 8
        .word   Nop             ;8
        .word   Inc_Min_Rows    ;9   P1
        .word   Srv_M1          ;10  Min 1
        .word   Srv_M2          ;11  Min 2
        .word   Srv_M4          ;12  Min 4
        .word   Srv_M8          ;13  Min 8
        .word   Inc_Min10       ;14
        .word   Srv_M10         ;15  Min 10
        .word   Srv_M20         ;16  Min 20
        .word   Srv_M40         ;17  Min 40
        .word   Chk_Min         ;18
        .word   Inc_Hrs         ;19  P2
        .word   Srv_H1          ;20  Hrs 1
        .word   Srv_H2          ;21  Hrs 2
        .word   Srv_H4          ;22  Hrs 4
        .word   Srv_H8          ;23  Hrs 8
        .word   Nop             ;24
        .word   Srv_H10         ;25  Hrs 10
        .word   Srv_H20         ;26  Hrs 20
        .word   Chk_Hrs         ;27
        .word   Nop             ;28
        .word   Nop             ;29  P3
        .word   Nop             ;30  Days 1
        .word   Nop             ;31  Days 2
        .word   Nop             ;32  Days 4
        .word   Nop             ;33  Days 8
        .word   Nop             ;34
        .word   Nop             ;35  Days 10
        .word   Nop             ;36  Days 20
        .word   Nop             ;37  Days 40
        .word   Nop             ;38  Days 80
        .word   Nop             ;39  P4
        .word   Nop             ;40  Days 100
        .word   Nop             ;41  Days 200
        .word   Nop             ;42
        .word   Nop             ;43
        .word   Nop             ;44
        .word   Calc_status     ;45
        .word   Nop             ;46
        .word   Nop             ;47
        .word   Calc_clk_error  ;48
        .word   Nop             ;49  P5
        .word   Nop             ;50  DUT+-
        .word   Nop             ;51  Year 10
        .word   Nop             ;52  Year 20
        .word   Nop             ;53  Year 40
        .word   Nop             ;54  Year 80
        .word   Srv_DST         ;55  DST1
        .word   Nop             ;56  0.1 DUT
        .word   Nop             ;57  0.2 DUT
        .word   Nop             ;58  0.4 DUT
        .word   Nop             ;59  P6
;
```

The routines “Integrate_Array()” and “Integrate_bit()” perform the tasks of placing new bit data in the data arrays. “Inc_Min_Rows( )” is a routine that increments the BCD data bits in all ten arrays for the units digit. This is done every minute to maintain the correct time phase of each of the ten possible minute values. “Calc_bits()” is a function to

calculate the BCD digit value from the bit data and also determine the total bit energy a given data array.

The format for the bit data arrays is:

0	1	2	3	4	5	6	n+2
value	Bit0	Bit1	Bit2	Bit3	Bit4	Bitn	energy sum

The routines “Chk_Min( )” and “Chk_Hrs( )” monitor the progress of the time data acquisition and set the main time variables accordingly.

The routine “Service_Sec()” is called every second to perform the overall time of day time keeping by incrementing the time of day variables “Sec_count”, “Min_count”, and “Hr_count” .

The routine “Calc_clk_error( )” is called every minute and calculates the internal clock error and determines how much to compensate the “Sec_pos” timer.

The front panel LED’s are used to show the progress of time acquisition. After time has been determined, the LED’s sequentially flash in a pendulum fashion. Timing of the LED’s is done with a lookup table that is called from the 5 mSec service loop.

TICSOUND.ASM Module

This module is responsible for creating the sound affects for the cuckoo clock. The method used is called FM Synthesis.

Essentially, FM is fast vibrato. When a vibrato rate moves into the audio range, and its depth is well in excess of that generally used in vibrato, the effect - instead of being one of fast up and down repeated glissandi, sliding above and below a base frequency - is to distort the waveform of the modulated oscillator ("carrier"), producing sidebands above and below the base frequency. Sidebands will either form harmonic or inharmonic relationships with the base (ie. the carrier) frequency. The positioning of sidebands is a function of the carrier: modulator ratio, and their number and amplitude vary in proportion to the amplitude of the modulator (ie. the depth of modulation). While the precise relationship between the sidebands, and the carrier:modulator ratio and the amplitude of the modulator (modulation index), can be determined by the use of Bessel functions, in general, it is reasonable to say that the number of sidebands produced on either side of the carrier will be equal to the modulation index plus 2. Further, the position (ie. frequency) of the sidebands will follow the basic rule:

carrier-frequency ± (k * modulator-frequency)

where k is the order of the sideband and generally ranges from 0 to the modulation index + 2

The formula for FM,

$\cos(w_c t + B \sin w_m t)$

where the "m" stands for "modulator" and the B is the "modulation index".
With a bunch of boring math, the expression can be expanded into it's spectrum of harmonics:

$$\cos(w_c t + B \sin w_m t) =$$
$$J0(B) \cos w_c t$$
$$- J1(B)[\cos(w_c - w_m)t - \cos(w_c + w_m)t]$$
$$+ J2(B)[\cos(w_c - 2\,w_m)t + \cos(w_c + 2\,w_m)t]$$
$$- J3(B)[\cos(w_c - 3\,w_m)t - \cos(w_c + 3\,w_m)t] + \ldots$$

Where the J's refer to the Bessel functions. The spectrum is made up of a "carrier" at w_c and symmetrically placed sidebands separated by w_m. The amplitudes follow the Bessel functions.

By changing the amplitude envelope and modulation index over time, various sound affects can be generated.
More details can be found at the following URL:
<http://ccrma-www.stanford.edu/CCRMA/Software/clm/compmus/clm-tutorials/fm.html>
The cuckoo clock's implementation uses a table containing the amplitude and modulation index envelope to generate each sound. In order to save space, the slope

of the envelope is stored rather than the absolute value making a linear piece wise approximation of a sound envelope very compact. Three independent FM generators are needed in order to create a tubular bell(gong) sound. The three generators are summed together and then sent to the D/A. The sound generator table format is as follows:

```
;***********************   SOUND GENERATOR TABLES ***************************&
; FM generator initial data structure entry format:
; NCO fequency rate increment = SampleFreq/65536 = 10000/65536 = .15258789 Hz
; The first 24 words are used for the initial FM generator values.
; Each of the three FM generators are initialized with 8 words
; in the following format:
;  [0] --->  modulation phz increment value
;  [1] --->  modulation phz accumulator
;  [2] --->  modulation index slope value
;  [3] --->  modulation index
;  [4] --->  carrier phz increment value(.15258789 Hz increments)
;  [5] --->  carrier phz accumulator
;  [6] --->  carrier amplitude slope value
;  [7] --->  carrier amplitude
; After the 24 words of initialization data, groups of 7 data words are
; used to define when and what parameters to change during the sounding
; of the generators.  The format for the 7 data words is:
;  [0] --->  When the change should occur(in 100uSec increments)
;  [1] --->  new FM gen 1 modulation index slope value
;  [2] --->  new FM gen 1 carrier amplitude slope value
;  [3] --->  new FM gen 2 modulation index slope value
;  [4] --->  new FM gen 2 carrier amplitude slope value
;  [5] --->  new FM gen 3 modulation index slope value
;  [6] --->  new FM gen 3 carrier amplitude slope value
```

The real trick is determining the frequencies and envelopes for the desired sound affect. Fortunately the sounds used were obtained from examples found in various documents on the WEB. The cuckoo sound was designed from a wave file of a Three Stooge's episode where Curly gets bonked in the head with the resulting desired cuckoo sound!

The routine "Start_Sound()" is called to initiate a sound. It is called with a pointer to the desired sound table.
The routine "Service_sound()" reads the sound table and obtains the next set of parameters to give to the three FM generators.
The routine "Calc_sample(AR0)" calculates a new sample of a single FM generator data structure pointed to by AR0.

A state machine is implemented that dispatches the various sounds depending on the current time.

```
;void Sound_idle_state()
;{
;   if( Flags.TIME_OK ){
;      if( Hr_count != Prev_hr ){
;         Prev_hr = Hr_count;
;         Sound_Vector = Sound_hr_state;
;      }else{
;         if( Min_count != Prev_min ){
;            Prev_min = Min_count;
;            switch( Min_count ){
;               case 15:
;                  Sound_Vector = Sound_15_state;
;                  break;
;               case 30:
;                  Sound_Vector = Sound_30_state;
;                  break;
;               case 45:
;                  Sound_Vector = Sound_45_state;
;                  break
;               default:
;                  break;
;            }
;         }
;      }
;   }
;   if( (Sec_count & BIT0 ){             // tick or tock every
;      Sound_Vector = Sound_tick_state; // second
;   else
;      Sound_Vector = Sound_tock_state;
;   }
;}
```

TICUART.ASM Module

This module converts the time of day into ASCII strings and squirts it out the UART every second.
Two ASCII strings are made. One is used to display status information before the clock has acquired the time data. The second one contains the actual time of day formatted information.

The function “itoa( )” converts a value from 0 to 99 into an 2 character ASCII string and places it into the main output string.
The routines “Calc_Sec()”,“Calc_Min()”, and “Calc_Hr()” convert the time data into the proper ASCII representation. “Calc_Hr()” has additional logic to display either local time compensated for Daylight savings time or GMT time in either 12 or 24 hour format. The 12 hour format displays AM or PM as a suffix.

The DTR and RTS lines on the serial port are toggled at a 1 Hz rate out of phase with each other.

TIC Clock Test Method

Several methods were employed in debugging and verifying the clock design. The primary measurement tool was an oscilloscope. For measuring timing, digital outputs were used such as LED ports or TNC port bits. For measuring signal data, the D/A channel of the DSP-93 was used to output various test points. A Telulex SG-100 signal generator was useful in evaluating the filters and detector code. For time data testing, WWV signals were used for tweaking and evaluating the TIC1 performance.

CLOCK PERFORMANCE

The processor load was roughly measured by measuring the peak and average time it took to service all four software modules in the main code loop. The minimum time around the loop was 9.2 uSec. Peak processing time around the loop while receiving and sounding, was around 95 uSec. This means the Sample_Que probably never gets even one sample behind. The average time around the loop was about 50 uSec. This means the processor is running about 50% of a full load at the present sample period of 100uSec. Code size takes about 4.1K of program space.

Areas for Improvement:

Acquire time could be improved by integrating analog bit values instead of the +/-1 discrete values. Determining energy thresholds would be more difficult but under strong signals, the lock time could be reduced to a few minutes.

Ideas for the Future:

None. This project has already taken up too much time.(pardon the pun)

References

- Marvin E. Frerking, “Digital Signal Processing in Communication Systems”
- K. Sam Shanmugam, “Digital and Analog Communication Systems”
- Texas Instruments, “TMS320C2x Users Guide”
- Texas Instruments, “Digital Signal Processing Applications with the TMS320 Family Theory, Algorithms, and Implementations” Vol. 2
- Nicky Hind, “Frequency Modulation Synthesis (FM)”
- John Chowning, “Frequency Modulation and some CLM examples”
- Dave Mills, “Precision Radio Clock for Station WWV/H Transmissions”
- NIST, <http://www.boulder.nist.gov/timefreq/pubs/sp432/s_wwv.htm>