A Cheap SDR Loran-C frequency receiver

(Formatted Thu Jan 8 09:41:03 UTC 2009)

Poul-Henning Kamp

<phk@FreeBSD.org>
Den Andensidste Viking
Herluf Trollesvej 3
DK-4200 Slagelse
Denmark

ABSTRACT

Loran-C radio-navigation signals are broadcast in most of the northern hemisphere and offer a solid alternative to GPS disciplined frequency standards, but a regrettable lack of affordable receivers means that Loran-C sees very little use for this purpose.

This article describes a simple, cheap and efficient Loran-C frequency receiver for such purposes.

The three main components of the receiver is a homebuilt loop antenna, the TAPR/N8UR "ClockBlock" and the OliMex.com ADUC-P7026 microcontroller prototype card, for a total cost well under $200.

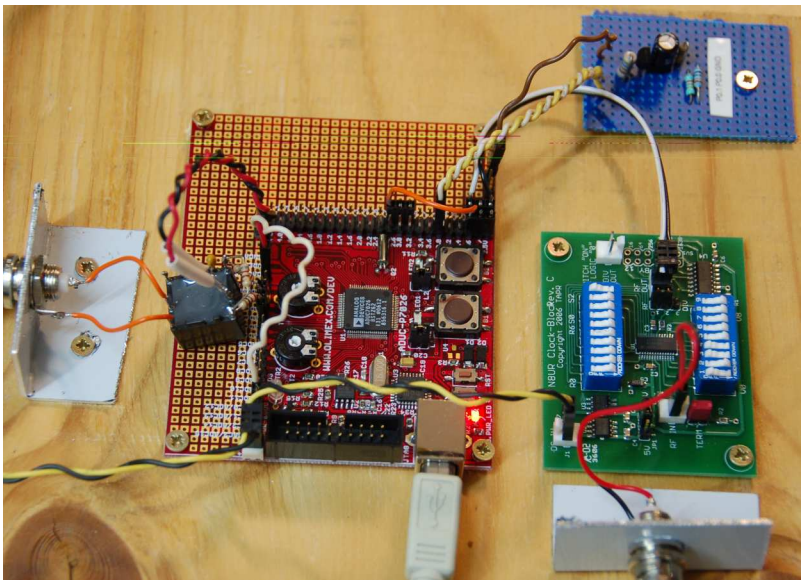

Figure 1. The complete receiver.

1. Introduction

LORAN-C signals, like GPS signals, offer wireless access to very stable and calibrated, and thus expensive, frequency sources and can therefore be used to steer cheaper oscillators to correct frequency.

Despite stern warnings to the contrary, the world today is more or less entirely frequency locked to the GPS satellites because that is the most convenient and cheap technology available.

This little project attempts to show that an equally cheap solution can be made based on the Loran-C signals, which are broadcast over most of the northeren hemishere.

Compared to GPS, Loran-C signals are just about as different as can be, where GPS satellits transmit their signals with only 50W using 1.2 GHz microwaves from 20,000 km above the Earth, Loran-C transmits are 200 meter tall steel towers which transmit several hundred kW at a frequency of 100 kHz.

This makes Loran-C the perfect backup for GPS, since there are almost no overlap between threaths to the two systems.

This paper documents what is clearly a "work in progress", and as time and energy permits, I will update both this paper and the source-code that goes with it.

Poul-Henning Kamp

2. Hardware

The entire reason for this article is that a new breed of microcontroller combines a real 32bit CPU with fast analog/digital converters on a single chip.

Amongst these chips my fancy took to Analog Devices ADuC7026.

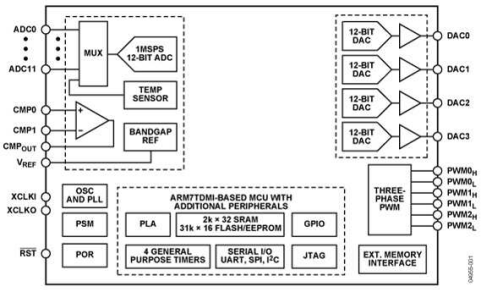

Figure 2. ADuC7026 block diagram

For one thing, Analog Devices are experts at the analog/digital border, and the specifications on the ADC in the ADUC is much tighter than on on competing chips.

Furthermore the ADC offers a very convenient fully differential input mode and is not picky about the ADC reference voltage.

There are also some downsides: The chip needs a 42 MHz clock frequency to run the ADC at 1 megasample per second, and it only has 8kilobytes of RAM.

A very interesting feature is the built in "programmable logic array", which makes it possible to totally avoid external glue-logic in our case.

The main market for the ADuC7026 seems to be the variable frequency motor control market, and certain on-chip facilities are less than well thought out and documented.

One example of this is that the internal timers barely have any connection to the outside and that trigger conditions have unexplained systematic delays.

As it transpires, there is a way to hook things up in a way we can use.

2.1. The 42MHz clock frequency

The 42MHz clock frequency is much less of a problem than it sounds, thanks to John Ackermann, aka N8UR, we can simply pick up a "ClockBlock" from TAPR.org, set the jumpers correctly and get 42MHz out of almost any input frequency we care to use.

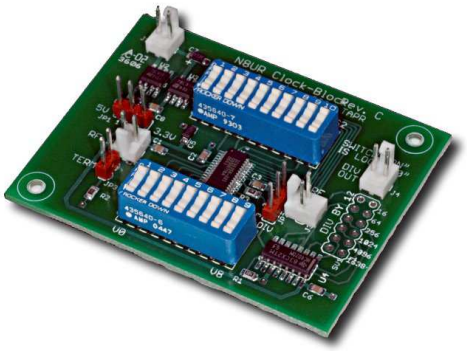

Figure 3. TAPR.org *ClockBlock* card

It is important to jumper the ClockBlock for 3.3V so it does not overdrive the ADuC7026 clock input.

The calculation of correct jumper settings is semi-complicated, and by far the easiest way is to use the web-page IDT provides:

```
http://timing.idt.com/
    calculators/ics525inputForm.html
```

For your convenience, here are three typical jumper settings:

F_{in} =	1MHz	5MHz	10MHz
S0	0	0	0
S1	0	0	0
S2	1	1	1
R0	0	0	0
R1	0	0	0
R2	0	0	0
R3	0	0	0
R4	0	0	0
R5	0	0	0
R6	0	0	0
V0	0	0	1
V1	1	1	0
V2	0	0	1
V3	1	0	1
V4	0	0	0
V5	0	1	0
V6	1	0	0
V7	1	0	0
V8	0	0	0

Figure 4. ClockBlock jumper settings

2.2. The ADuC7026 microcontroller

Being mostly of the software persuation, I find the prospect of soldering 80 pin SMD packages something to be avoid if reasonably possible.

Fortunately my favourite Bulgarian electronics pusher, OliMex.com, has an ADUC-P7026 prototype card which is perfect for this project.

In the USA, you can purchase it from Sparkfun.com.

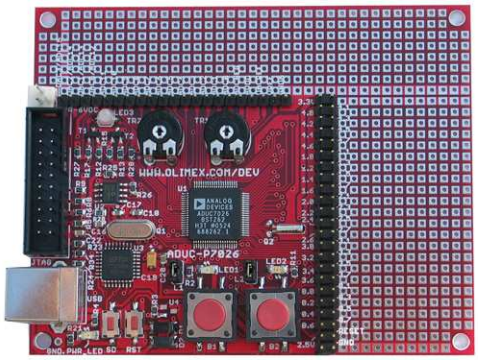

Figure 5. Olimex.com *ADUC-P7026* card

2.3. The Antenna

I do not really consider the antenna part of the project, because I reused a very simple loop antenna I built for a previous round of Loran-C experiments, based loosely on a diagram I found on the website `VLF.it`, using an AD797 operational amplifier.

Figure 6. $20 homebrew loop antenna

It goes without saying that I have not spent a lot of time on antennas, but roughly speaking, the requirements for the antenna input signal are something like:

- ☐ A Relatively flat passband from at least 70 to 130 kHz.
- ☐ Not too much signal above 300-400 kHz.
- ☐ At least 100mV peak-to-peak Loran-C signal.
- ☐ No more than 3V peak-to-peak total output signal.

2.4. The Antenna Balun

The ADC in the ADUC 7026 microcontroller can operate in a number of modes, but by far the most convenient for our purposes is the truly differential mode.

A differential CMOS ADC can be conveniently driven by a signal transformator of the kind found in old ADSL or ISDN equipment.

One particular nice thing about this approach is that a transformator has offset voltage.

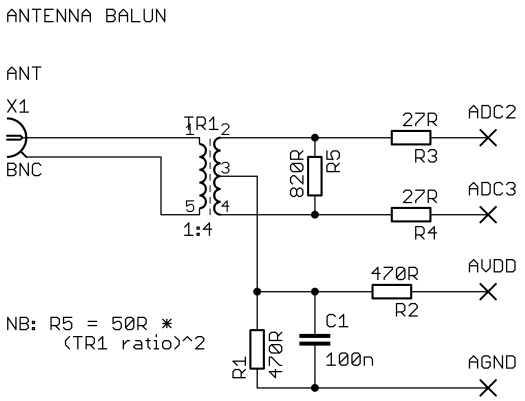

Figure 7. Antenna Balun Schematic

To be honest, this balun consists of whatever I found in my junkbox and is not optimized in any way shape or form.

The transformer I found is a 1:2 trasnformer from an ISDN NT box, where both sides have center taps. I feed only one half of the input side so it looks like a 1:4 with secondary center-tap.

Notice that R5 should be chosen as the input impedance multiplied by the square of the transformators ratio:

$$R5 = Z_{in} \cdot N^2$$

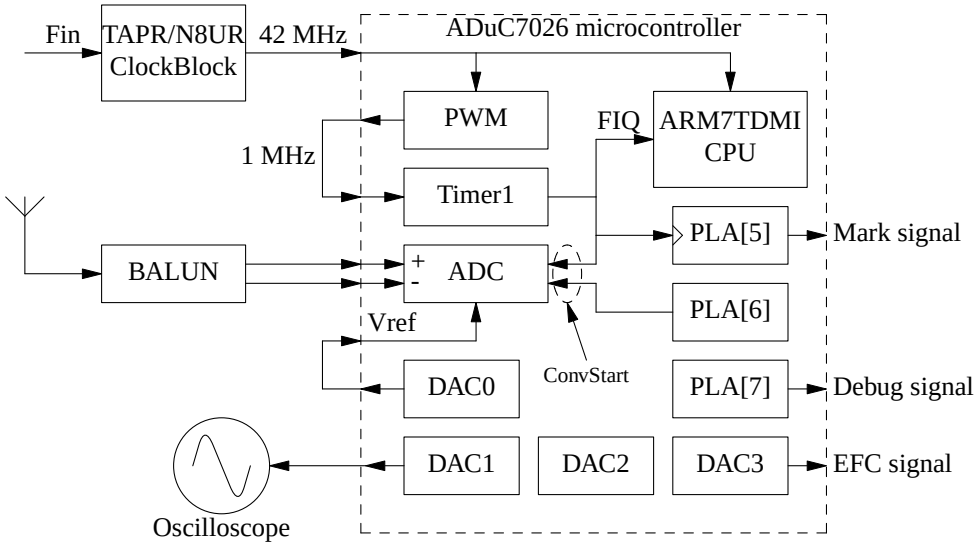

Figure 8. Blockdiagram of the receiver hardware.

2.5. Connecting it all

That is really all for the hardware, now we just need to connect it together:

- □ Jumper the ClockBlock for 3.3V supply.
- □ Connect the ClockBlock output to pin P0.7 on the ADUC-P7026.
- □ Connect your chosen frequency standard to the ClockBlock.
- □ Connect pin P2.7 ($PWM1_L$) to P0.6 (*T1*) on the ADUC-P7026 to establish a connection from the PWM output to the Timer1 input.
- □ Connect pin XXX to XXX on the ADUC-P7026 to establish a connection from DACXXX output to the ADC's VREF input.
- □ Connect the differential antenna signal to pin ADC1 and ADC2, remember that they both must stay between A_{gnd} and A_{vdd} at all times.
- □ Find a suitable power supply for both the cards, it should be at least 6VDC to avoid drawing power from the USB port, and needs to supply about 100mA total.
- □ Connect your computer to the USB port on the ADUC-P7206 card.

2.6. Available output signals

The following output signals are available for regulation and debug use.

- □ P2.1 (*PLAO[6]*) "Mark" output. The raising edge marks the start of measurements in the A-code GRI interval. The software reports how much later than this flank the 3rd zero crossing is found.
- □ P2.2 (*PLAO[7]*) "Debug" output. This pin flips low whenever and as long as the fast interrupt routine runs.
- □ P0.5 (ADC_{busy}) This signal is logic high while the ADC is performing a conversion
- □ DACXXX Analog replica of the averaged signal. Show this on your oscilloscope, use the "mark" output as timebase trigger.

2.7. Bootloader trigger circuit

The ADUC-7026 has a very convenient serial boot-loader which I use to download the firmware to the internal flash memory.

Unfortunately, there is no way to enter the boot-loader from software, it can only be triggered if P0.0 is logical low during a reset.

This little circuit implements a crude mono-table timer which can hold P0.0 down for us, while we trigger a reset from software, thus saving a trip down to the lab.

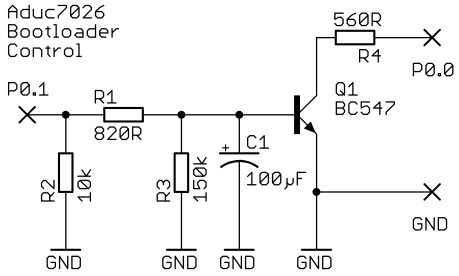

Figure 9. Bootstrap circuit

3. Loran-C signals

A Loran-C navigation chain consists of a master transmitter and from one to five slave transmitters, all broadcasting on the same "GRI", Group Repetition Interval.

A Physical transmitting station can participate in multiple chains, although for reasons of time-contention, typically it will only participate in two chains.

In our case, we are interested only in a single transmitter in a single chain, typically the nearest and strongest signal at the location of reception, although there is nothing preventing this construction from being used for more ambitious reception projects.

3.1. What the station transmits

There are many ways to describe the Loran-C signal, but for our analysis of how and why this receiver works, we will treat it as three components which are convoluted to give the actual on-air signal.

3.1.1. The GRI periodic function

This is basically a pulse generator which emits a pulse every GRI * 10 μseconds, so that for instance the Sylt chain with GRI 7499 has a period of 0.07499 seconds.

A very important feature of the GRIs used, is that they do not result in periods that are integral multiples of 1 msecond, so signals from CW radio transmitters will average out, allowing us to use essentially no other frequency domain filtering.

3.1.2. The signal code function

This is where much of the noise resistance of the Loran-C signal structure comes from: in alternating GRI periods two different pulse-trains are used for each of Master and Slave stations.

Whenever the GRI periodic function triggers a transmission eight or nine pulses will be transmitted with 1 msec interval, with polarity according to the following two figures.

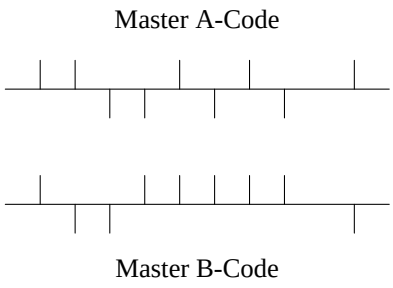

Figure 10. Master Signal Codes

In addition to the frequency filtering provided by the GRI averaging, the non-periodic nature of these codes provide significant improvements in S/N ratio.

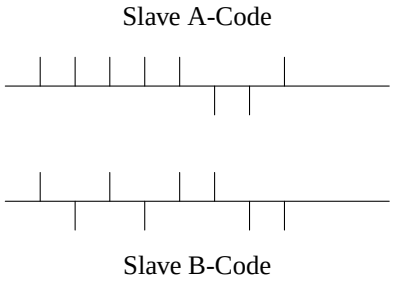

Figure 11. Slave Signal Codes

One important feature of both codes is that if the A and B interval codes are summed up, the odd numbered pulses cancel out leaving only four signals, spaced 2 msec apart.

3.1.3. The Radio Frequency Pulse function

The final step of the modulation is that each pulse is transmitted as

$$f(t) = \sin\left(\frac{2t\pi}{10}\right) t^2 \exp^{-2t/65}$$

where t has units of μseconds.

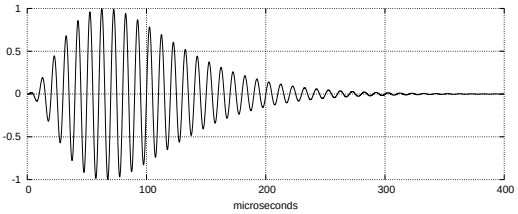

Figure 12. Loran pulse

In reality it is only the first part of the pulse-shape which is controlled, and in particular, the reference point is the 3rd positive zero-crossing of the signal

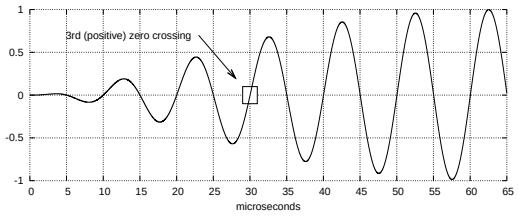

Figure 13. Loran pulse front

When we combine all these components, the result is something like this:

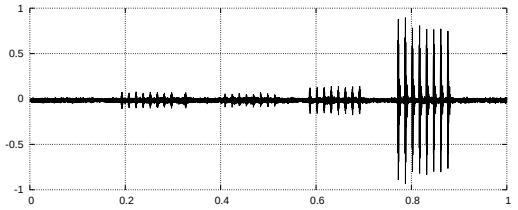

Figure 14. 6731 Lessay chain

All four stations in the Lessay chain are visible in this plot, first the master in Lessay, notice the "extra" 9th pulse 2 msec after the 8th, the follows the Xray slave in Soustons, the Yankee slave in Rugby and finally the Zulu slave on the island of Sylt.

Knowing the exact geographical locations of these four transmitters, and the time interval from the master transmission until the slaves transmit their pulses, we could calculate our geographic position from the measured time-intervals between these signals.

4. Reception

So, how do we find and lock onto the Loran-C signal with only 8 kilobytes of RAM ?

The brute-force method would be to simply average all samples over the FRI interval, that would allow us to identify all signals in the chain, tell A and B codes apart, locate the 3rd zero-crossing and track each signal.

Unfortunately, this would require up to $2 \cdot 9999 \cdot 10 = 199980$ averaging buckets.

Even if only one bit is used per bucket, this amounts to 24.5 kilobytes, more than three times the amount of RAM we have available.

We can get rid of the factor two by averaging only over the GRI interval, and provided we rotate every other GRI period one millisecond, we can still tell the four signal codes apart, at the expense of recognizing two energy levels in the pulse patterns.

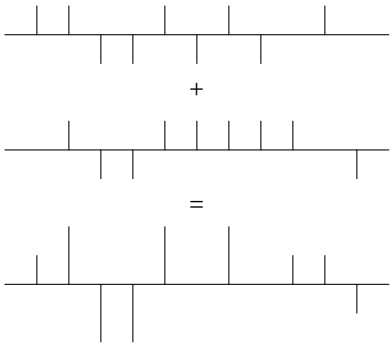

Figure 15. delayed master code average pattern

Unfortunately, that still requires more than 12 kilobytes of RAM.

It is possible to implement both of these versions of the algorithm by looking only at a small window of the GRI or FRI period at a time.

Since we realistically need 16, or preferably 32 bits, per bucket, and at best can use around 6 of the 8 kilobytes, this would result in at least 33 windows, in practice more because of necessary overlapping, each of which must be integrated for some time, before a signal may be appearant.

Instead we this narrow scan, single phase approach, we opt for a widescan, multi phase approach.

5. Phase Zero - Locating the strongest signal

Our first phase must locate the strongest signal in the chain, typically the transmitter closest to the receiving antenna.

When averaged over the GRI period, both the master and slave codes become four pulses separated by $2ms$ and our goal is simply to find the strongest one of these.

We do not care at this point, if it is a master or slave signal or where the A or B codes are in time, the next phase will figure that out.

Assuming we have 1000 buckets of 32 bits available, we can sample the GRI interval every $100\mu s$, and in doing so, we would likely miss even very strong signals: We need a sampling frequency which is not a divisor in the 100 kHz carrier frequency.

Since our goal is to just find the rough location of the strongest signal in the GRI, we are not even constrained to sample at rates that divide into the $1ms$ pulse spacing, and in fact get better results by not doing so.

Experiments and simulations has shown that if we average every $114\mu s$, every 114th sample, over the GRI period, we can expect to catch at least 30% of the peak amplitude of the strongest signal, and need only $99990\mu s/114 = 887$ buckets.

5.1. Tek4014 view of phase zero

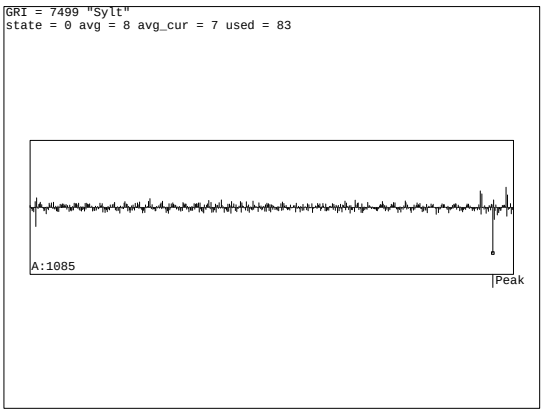

Figure 16. Tek4014 view of phase 0

This is how the tek4014 window looks in phase zero, we see the averaging buckets plotted, and the peak signal is marked.

In the lower left corner of the box is the amplitude of the signal in ADC units.

Notice that we do in fact quite clearly see all four peaks in the averaged signal, but one of them is significantly taller than the other three.

5.2. Hardware view of phase zero

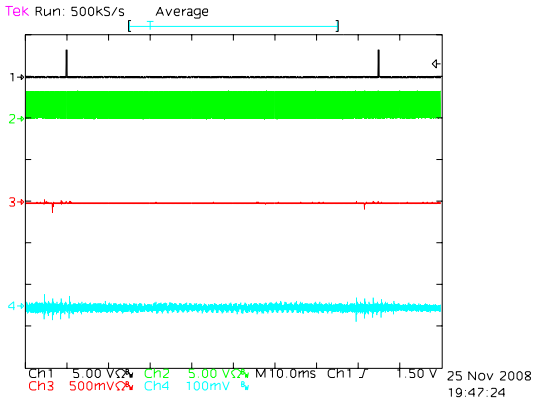

Figure 17. Phase 0 GRI wide signals

Trace 1 (black) is the marker output which is used to trigger the oscilloscope.

Trace 2 (green) is the debug output.

Trace 3 (red) is the DAC1 output, this is the same data which is plotted in the TEK4014 window.

Trace 4 (cyan) is the antenna input signal, where we can spot the LORAN pulses, thanks to a bit of averaging in the oscilloscope.

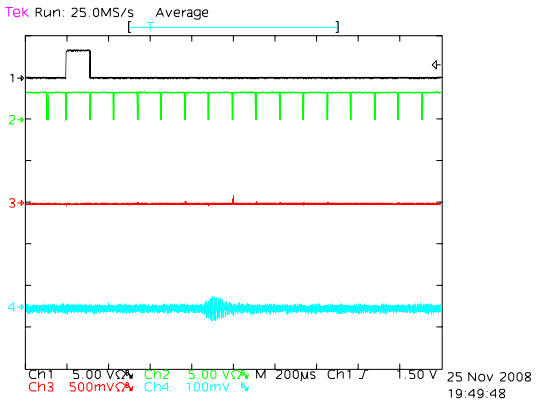

Figure 18. Phase 0 zoomed signals.

Zooming in, we see a single pulse of the loran signal in trace 4. Trace 2 shows the sample distance of 114 microseconds and trace 3 shows that we almost managed to miss this loran pulse entirely between two samples.

6. Phase One - Identifying the code

The peak signal from phase 1 must by definition be within 57 samples of the true peak value, of one of the four peaks, 2 msec apart, resulting from the GRI summing.

Now we need to find out if this is a master or slave station, and to identify which half of the FRI has the A and B code.

If we caught the last of the four peaks, the code starts 6 msec ahead of this point, if we caught the first of the four peaks, it spans another 9 msec after this point, so all in all, there are 16 windows we need to check for the precense of pulses.

We use a repetition frequency of 2 * GRI, sometimes called "FRI" for Frame Repetition Rate, in order to not add the A and B codes together.

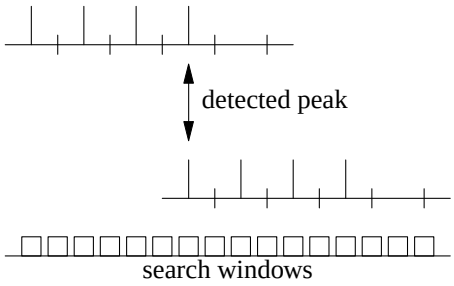

Figure 19. mode1 code windows

Using no more than the 876 integers used in phase zero, each bucket can get 54 integers and if we space them 13 samples apart, each bucket will cover 702 μseconds.

Again, 13 is a good number because it will make the sample points "wander" over the 100 kHz period of the pulses.

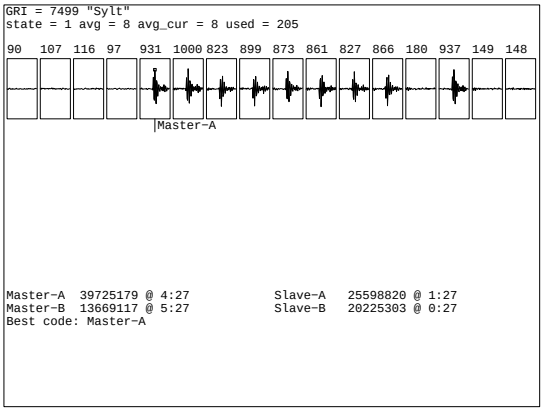

Figure 20. Tek4014 view of phase 1

The tek4014 view shows the 16 windows and shows the starting point of the strongest code in them.

Above each window is the energy content of that window, in thousands of the maximum found in any window. Due to timing effects, it is quite likely that one or two windows show numbers slightly above 1000. Presently, the energylevels is not used for code identification.

Below the windows are statistics for each of the best match for each of the four possible codes.

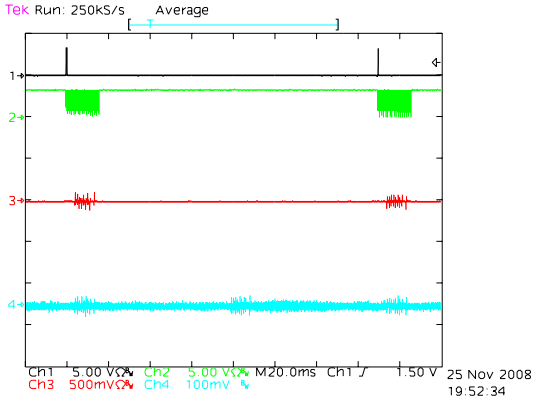

Figure 21. Phase 1 FRI wide signals

This is the hardware view, showing a full FRI interval.

Notice in trace 4, that we only sample every other pulse group.

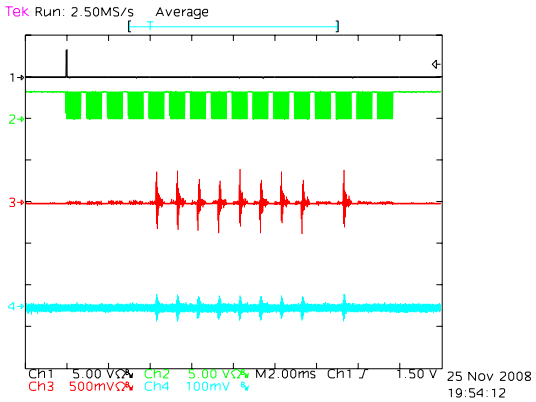

Figure 22. Phase 1 pulse group

Zooming in on the pulse group, we can see in trace 2 how each bucket is sampled and in trace 3 the resulting waveform.

Notice that the pulse shape is not fully recovered at this point, we only sample every 13

microseconds, but that is plenty to identify the code.

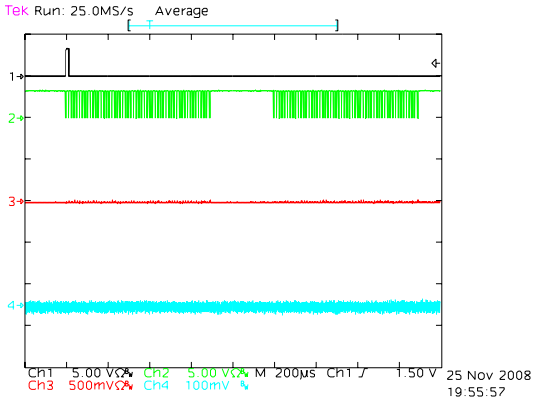

Figure 23. Phase 1 single pulse

Here we are zoomed into just two pulse windows.

7. Phase Two - Locking on to the signal

Now we have all the timing informaton we need to start integrating the signal for good: we know where the maximum is of the first pulse in the code, and we know which code it is.

In this phase we integrate every ADC conversion in a 750 (XXX: check code) sample window around all the 16 or 18 pulses in the signal, into the same single bucket, taking care to invert the pulses per the code.

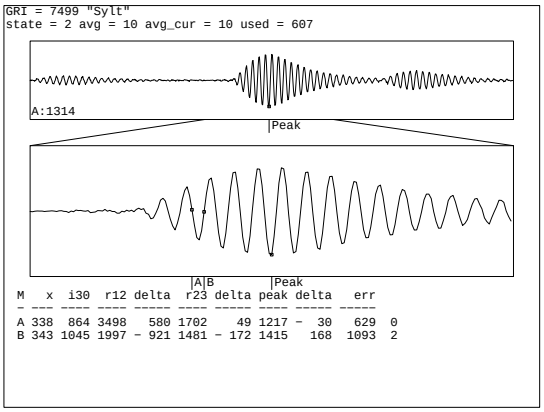

Figure 24. Tek4014 view of phase 2

The tek4014 view shows two views of our integration buffer, with the bottom one zoomed in on the peak value.

The various candidates for 3rd zero crossings are marked and their statistics given in numeric form below.

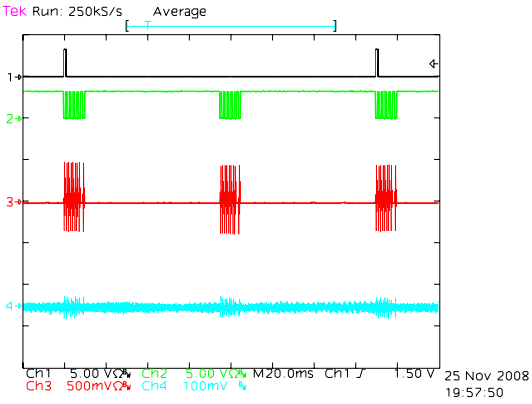

Figure 25. Phase 2 FRI view

The MARK pulses are spaced 2 * GRI apart now, and we can see how the signal is sampled twice in that period.

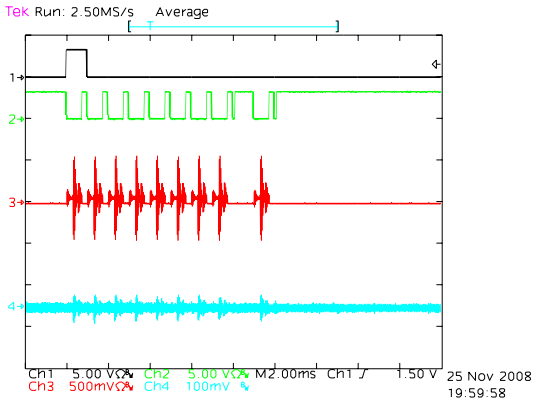

Figure 26. Phase 2 group view

Zooming in on the group, we can see th nine pulses of the master code being sampled.

Notice in trace 2, that there is a short burst of samples in the space between the 8th and 9th pulse, and after the 9th pulse. These samples are not technically necessary, but are planned for AGC use.

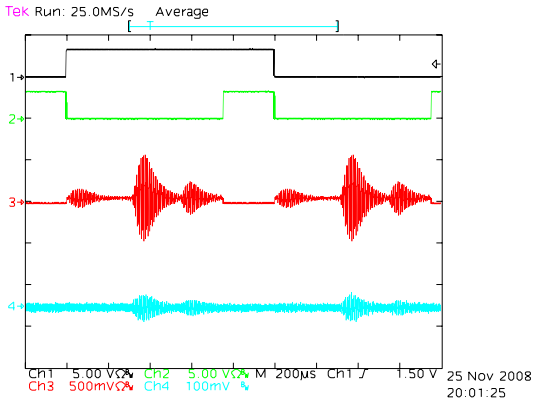

Figure 27. Phase 2 pulse view

Finally zooming in on a single pulse, we can see the full wave-shape of the Loran-C pulse, including the sky-wave echo.

The small "pre-echo" is currently unexplained.

7.1.

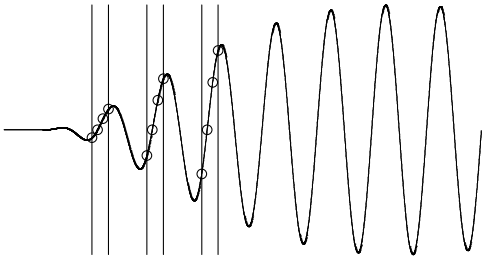

Figure 28. Finding 3rd zero

8. Phase Three - Tracking the 3rd zero-crossing.

In state 3 we have chosen the zero-crossing we want to track, and hold on to it to the best of our ability.

In practice we cannot simply nail a particular sample in the GRI window and track that, as the signal and the timebase may wander relative to each other.

We always use as tracking sample, the sample that is closest to the 3rd zero crossing.

Nominally the tracking point is sample 75 in the window, but we allow it to wander up to 10 samples in either direction, before we realign the sampling window to put it back at sample 75.

We interpolate the true zero crossing between the tracking sample and the two samples that surround it.

This is the Tek4014 display in phase three:

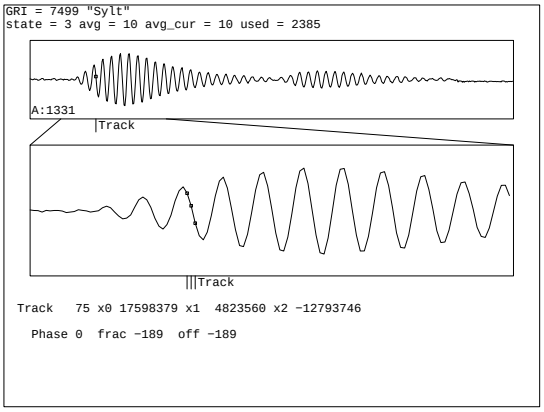

Figure 29. Tek4014 view of state 3

The top plot shows the entire sampling window and a marker for the tracking sample.

The bottom plot zooms in and marks both the tracking sample and the two neighboring samples.

Below the plots are the pertinent statistics, including calculated timing of the 3rd zero crossing.

The three samples which contribute to the interpolation of the zero crossing, are each filtered with a FIR bandpass filter:

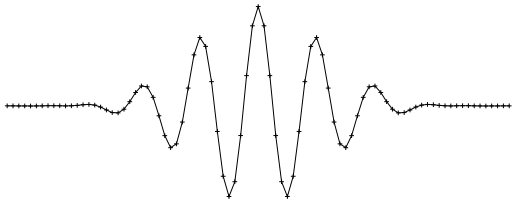

Figure 30. The FIR filter kernel

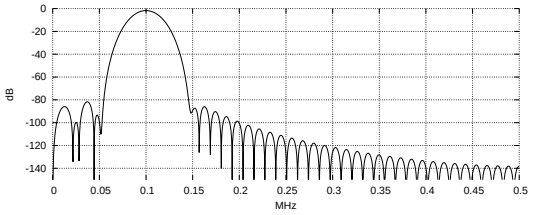

Figure 31. The FIR filter frequency response

The exact choice of filter has proven to be remarkably unimportant, but it does in fact reduce the noise in the three sample points considerably.

8.1. Interpolating the zero crossing

Since we want better resolution than the whole microseconds the sample rate provides, we need to interpolate the true zerocrossing from the sample values around it:

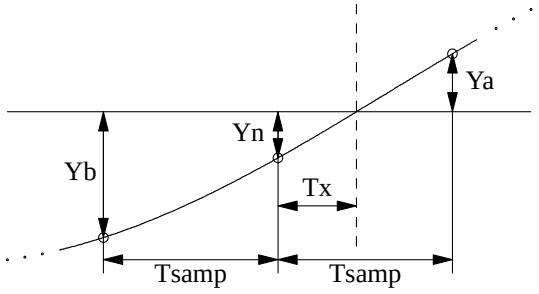

Figure 32. Interpolating the zero crossing

If we define Yn as the sample that has the lowest absolute value, closest to the 3rd zero-crossing, we can determine the range of *Tx* by solving:

$$Loran(x) = -Loran(x + 1\mu s) \quad x \in [29\mu s \cdots 30\mu s]$$

Doing so, we find $x \approx 29.664\mu s$.

It follows that the *Yn* sample must be located between $[29.664\mu s \cdots 30.664\mu s]$ and consequently that $Tx \in [-337ns \cdots 664ns]$.

The analytical solution for finding *Tx* given *Yb*, *Yn* and *Ya* is neither practically realizable with the limited amount of CPU and memory we have available, nor necessary.

Instead we calculate the FIR filtered signal at the sample nearest the zero-crossing (Fn), and the two samples right before (Fb) and right after (Fa) and find the ratio:

$$\frac{Fn}{Fa - Fb} \in [-0.2871 \cdots 0.2679]$$

Which we map to the corresponding:

$$Tx \rightarrow [-337ns \cdots 664ns]$$

Using a lookup table containing the precomputed conversion function:

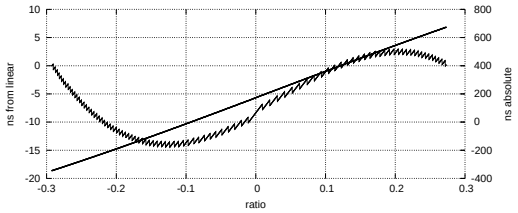

Figure 33. Interpolation non-linearity

The lookup table is built by the script `pulse_param.tcl` and lives in the source file `pulse_param.c`.

The script automatically reads in the FIR filter coefficients from the `fir.c` source file and determines how many entries the lookup table needs to have, to provide the specified resolution.

The table is extended 5% further than necessary in both end, to cater for edge effects and noise components.

For a resolution of nanoseconds, the table has 1111 entries.

8.2. Serial output data

```
agc     1324
avg     10
track   75
yb      22376693
yn      8627207
ya      - 8645207
Fyb     12611456
Fyn     2030392
Fya     -10263240
µs      105
Phase   0
frac    -185
off     -   185
#
```

Figure 34. Serial data stream

9. Performance

These are some of the first performance data collected, and they should therefore be taken with all sensible precaution.

The signal is 7499M, 205 km away.

The exponential averaging factor is 1/1024.

The timebase is a free-runing FRS10 Rb, the samples have been compensated for the randomly chosen, but deliberate 7.225e-10 frequency offset.

A phase measurement is recorded once per second using Timer2, from approx 2008-11-30 11:00 to 2008-12-01 11:00 UTC.

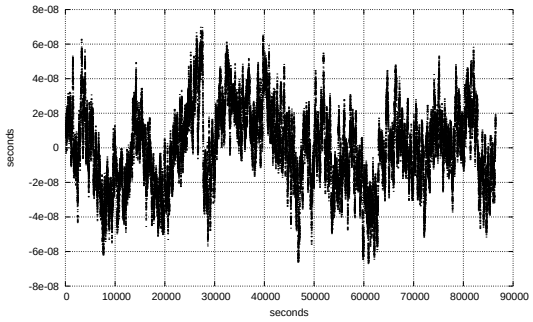

Figure 35. Raw offset samples

The standardeviation of the raw samples is 22.6 nanoseconds.

A histogram of the samples show a gaussian distribution, which seems well suited to more aggresive averaging.

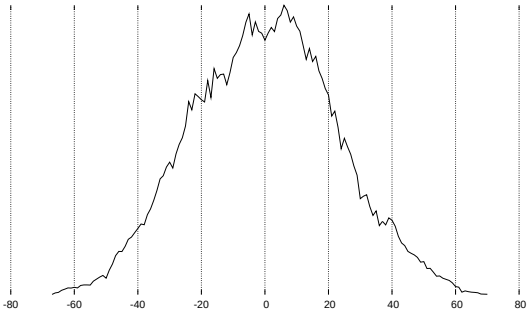

Figure 36. Offset value interval histogram

The modified allan variance is plotted below:

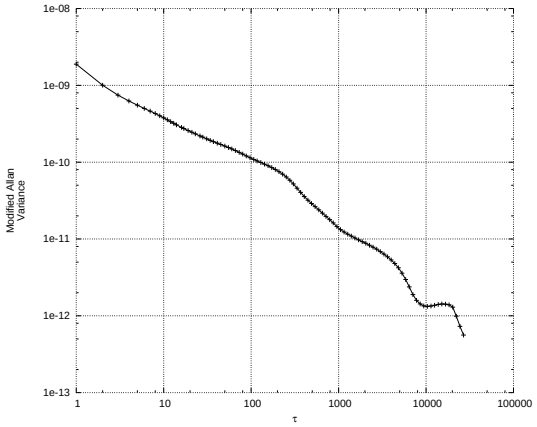

Figure 37. Modified allan Variance

10. Software

The software that goes with this article is writte as proof of concept, but is written such that experimentation and further development is possible.

10.1. The tricky assembler bits

The tricky bit of the software is the assembler coded fast interrupt routine in the file *crt0.S*.

I have written this code so that it does not encode any of the high-level logic, but instead mechanically walks a chain of structures that tell it what to measure and when.

For instance, when we scan the 6731 GRI in phase zero, the specification looks like this:

```
{
    mark         = 1;
    ptr          = ss->ptr;
    polarity     = "+";
    avg          = 6;
    cnt          = 1;
    timer_repeat = 114;
    repeat       = 590;
    timer_after  = 50;
    next         = this;
}
```

Figure 38. Phase zero specification

This specification tells the assembler code to pulse the MARK pin high when starting this specification, then 590 times take one ADC measurement 114 microseconds apart, then skip 50 microseconds and start over.

Since 590 * 114 + 50 = 67310 this will repeatedly scan the 6731 GRI signals.

The *polarity* member specifies that the samples all have positive phase, the *avg* specifies he exponential average time constant and the *next* member tells it what to do next (the same thing).

With none of few modifications, this scheme should also support reception of multiple stations in the same chain or even several stations from different chains.

10.2. The high level logic

The highlevel logic, such as it is, is written in C code in the *loran0.c* file.

Right now the code is semi-manual, controlled by single characters received on the serial/USB port, running at 115200 bps.

The following commands are recognized:

c This will present a menu with Loran-C chains from which you can choose one by entering the character in the [...].

```
---------------------------------
[@]  5543  Calcutta
[A]  5930  Canadian East Coast
[B]  5980  Russian-American
[C]  5990  Canadian West Coast
[D]  6042  Bombay
[E]  6731  Lessay
[F]  6780  China South Sea
[G]  7001  Bø
[H]  7030  Saudi Arabia South
[I]  7270  Newfoundland East Coast
[J]  7430  China North Sea
[K]  7499  Sylt
[L]  7950  Eastern Russia Chayka
[M]  7960  Gulf of Alaska
[N]  7980  Southeast U.S.
[O]  7990  Mediterranean Sea
[P]  8000  Western Russia
[Q]  8290  North Central U.S.
[R]  8390  China East Sea
[S]  8830  Saudi Arabia North
[T]  8930  North West Pacific
[U]  8970  Great Lakes
[V]  9007  Eiði
[W]  9610  South Central U.S.
[X]  9930  East Asia
[Y]  9940  U.S. West Coast
[Z]  9960  Northeast US
[[]  9990  North Pacific
---------------------------------
Please select chain:
```

Figure 39. Chain selection menu

t Make TEK4014 plot via serial/USB port

u Zoom in on curves in TEK4014 plots

d Zoom out on curves in TEK4014 plots

a Double the exponential average time constant.

b Half the exponential average time constant.

1 Skip to mode 1

2 Skip to mode 2

10.3. TEK4014 plotting

Before windows and mice, Tektronix produced a series of high quality graphical terminals, based on the storage-CRT principle.

These plotting instructions to these terminals became the de-facto format for plotting files and as a result, the original X11 "xterm" program offered, and still does, TEK4014 emulation.

If you connect to the serial/USB port using the X.org xterm program and press ’t’, then you will be rewarded with a nice plot as seen earlier in this paper.

If you use another terminal program which does not support TEK4014 plotting commands, you will get a blast of random ASCII characters.

10.4. Utility functions etc.

The other three C language files contains the code to configure the hardware, code to use the serial/USB port and a function which calculates

basic statistics for an array of integers.

The files divdi3.c, qdivrem.c and divsi3.S, are runtime support files for the GCC compiler.

10.5. Downloading firmware to the ADuC7026

You can either use a JTAG gadget or download the firmware via the serial/USB port.

Included in the source-code package is an "aduc" program I have written for that purpose.

10.6. Compiling the firmware

Cross-compiling is notoriously tricky, but the FreeBSD build enviroment makes it quite easy.

To build a arm cross-compiler + toolchain, simply do the following:

```
cd /usr/src
make toolchain \
    TARGET=arm TARGET_ARCH=arm
```

Figure 40. Building an ARM toolchain on FreeBSD

The Makefile knows how to find these tools under the /usr/obj.

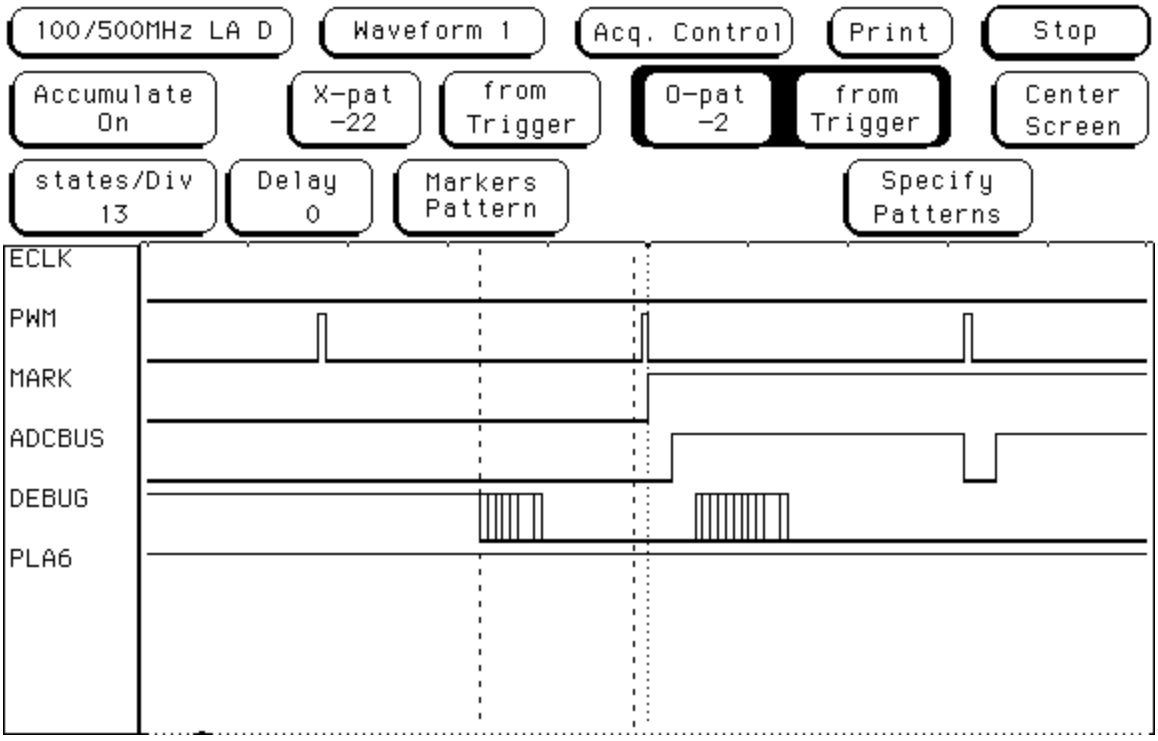

Figure 41. Protocol analyser trace

11. Timer1 unexplained

This is a confession: I have no idea exactly how Timer1 is supposed to work in the ADuC7026 chip, and the way I have observed it work baffles me, but the software manages to work around it.

The PWM section provides us with a 1MHz frequency which, through an external pin, becomes Timer1's clock signal.

In Fig XXX above this external signal is named PWM, which for this experiment, was give a very short "on" time.

The signal named "MARK" is from PLA[5], which is configured so the flip-flop is latched by Timer1's count-complete signal.

The signal named "ADCBUS" is the ADCbusy signal, which indicates that the ADC has started performing a conversion, and the ADC is also triggered by Timer1's count-complete signal.

So far, so good, it looks like Timer1 expires around the middle of the figure.

But now look at the "DEBUG" signal, which is programatically lowered as the first thing in the fast interrupt handler, then pulsed high after Timer1 has been reloaded.

The fast interrupt is *also* triggered by Timer1's count-complete signal, but this happens well before PLA[5] and the ADC receive the signal.

From the FIQ request is raised until the first instruction of the handler is executed takes at least five clock cycles, but it can take as much as 13 (XXX ?) cycles, depending which instruction the CPU was executing.

This is why the "DEBUG" trace shows multiple transititions: it is not deterministic in time.

But he mystery gets deeper here, because the previous PWM signal, presumably the one that counds down to one, is a fair bit further ahead of the DEBUG signal than 5-13 (XXX) clock cycles.

But it gets more mysterious still: The fast interrupt handler loads a value into the T1LD register right before the high pulse on the DEBUG signal (right of the dotted line).

If the leftmost PWM signal is the 1 count, then the central one is the 0 count, it would be reasonable to expect that loading N or N-1 into the T1LD register at this point, would cause the next Timer1 count complete event to happen N cycles

of the PWM signal later, starting with the one in the right hand side of the trace.

Experiementally I have found that I have to load N-3 to make that happen.

For this to make sense, the PWM edge that triggered the count down to zero must be just outside the papers left edge so that the central PWM pulse is number 3 in the new counting cycle.

If that is the case, then the chip applies approximately 3μs worth of metastability latching on Timer1s count-complete output, before it reaches the ARM7TDMI core and the ADC unit.

At the 42MHz clock, 3μ correspond pretty precisely to 128 clock cycles.

If that is the depth of the latch-line that resolves metastability, then I would pressume to think that it is sufficient deep.

Enquiring minds wants to know...