Simple Oscillators

Including high-frequency crystal excitation

Although internal timing loops in our chips can be calibrated to suffice for many application requirements, there are times when a more accurate time or frequency reference is demanded. In addition one may wish for a way to introduce time delays without incurring the 4.5 mW power consumption of a node running timing loops. In this App Note, we explore several different ways of using minimal external circuitry to accomplish these things, and will compare their energy costs.

Our first work in this area covered low frequency/low Q devices such as 32.768 kHz watch crystals or 10 MHz ceramic resonators, beginning in 2010 when our first prototype of the GA144 was available. This edition adds excitation of high frequency/high Q devices. Further low frequency work followed in 2013, and work at high frequencies began in Spring of 2015.

Contents

1. Problem Statement

Although internal timing loops in our chips can be calibrated to suffice for many application requirements, there are times when a more accurate time or frequency reference is demanded, such as when transmitting serial data to devices that demand precise timing. In addition one may wish for a way to introduce time delays without incurring the 4.5 mW power consumption of a node running timing loops. In this App Note, we explore several different ways of using minimal external circuitry to accomplish these things, and will compare their energy costs.

Our ideal model for this function was one node with some external device on a single GPIO pin. Our examples used minimal circuitry, such as a single resonant device connected between a single pin and ground. As far as we know, exciting and using an external resonant or R-C device under program control was something only GreenArrays had contemplated around 2010, and as of 2017 it appears we are still the only chip that can afford to do this sort of thing.

The *magic* in these methods comes from our uncommon I/O pin electrical characteristics; from our computers that are fast enough to implement *software defined I/O*, generically referred to as "bit banging", at uncommon speeds; from the tight synchronization our nodes can achieve with external events using asynchronous pin wake-up; and from an architecture that permits us the convenience of employing an entire computer to attend to the duties demanded at a single I/O pin.

In the following sections we explore the use of external resonant devices and R/C networks.

1.1 Test Platform

1.1.1 Hardware Modifications

For this work we have used a standard test board, one for the prototype chips and the EVB001 after we reached production. Most of the experimentation and measurement was done using the Target chip, to facilitate measuring power without any other activity of that chip. Hardware modifications consisted only of attaching the external circuitry of each type.

1.1.2 Dual Chip IDE Operations

For manual verification of connections to the target chip, and later for booting the two-chip platform, we use the standard dual-chip IDE capabilities built into arrayForth.

`bridge load` compiles a version configured for two chips with a default path 0 that can reach all nodes of the Target chip, with or without the polyFORTH virtual machine present on the Host chip. Extended node numbering is supported in the form **`cyyxx`** where **`c`** is zero-relative chip number; thus nodes 000 through 717 are on the Host chip, while nodes 10000 through 10717 are on the Target chip.

`talk` is then used to reset the host chip and program its node 708 for IDE operations.

span resets the Target chip and builds a transparent bridge in node 300 of each chip for carrying port communications between the chips, dedicating those nodes to this purpose until next reset.

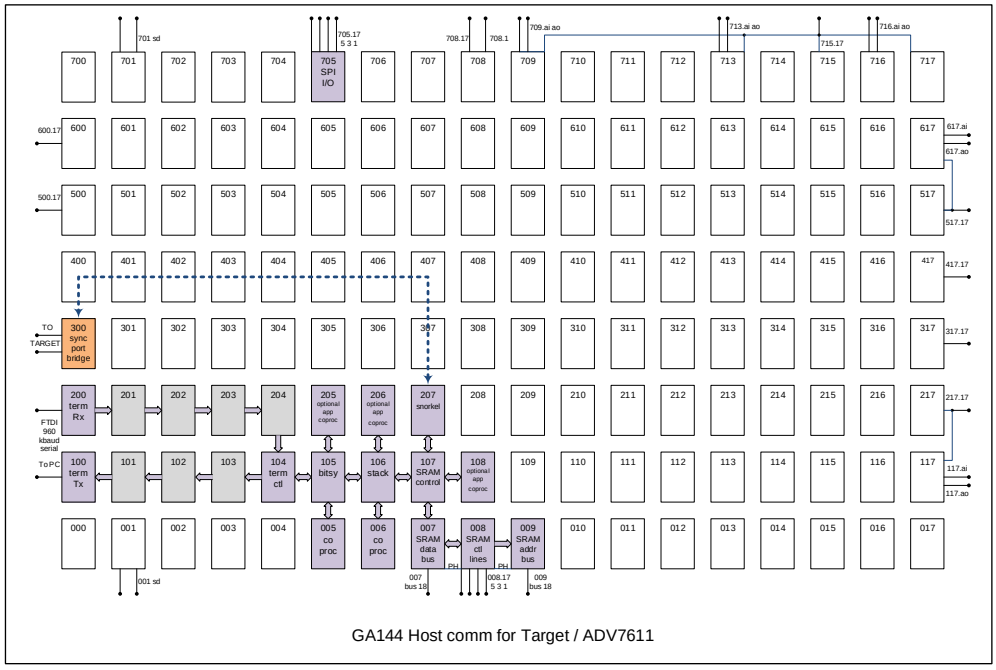
With the port bridges installed, the **`up`** ports of node 400 on each chip are logically connected as though they were a simple COM port. Port read/write communications, such as IDE, are basically transparent across this connection except that data transfers take 100 or more times longer, no flow control is supported, and polling of **`io`** by node 400 has limited usefulness. Node 300 will seem to be writing when a word sent by the other chip is waiting to be read, it will seem to be neither reading nor writing during serial data transmission in either direction, and it will seem to be reading at all other times regardless of the state of node 400 in the other chip. Path 2 is redefined to access most of the Host chip without interfering with the top row or left edge down to node 300, Path 0 is redefined to access all nodes of the Target chip, and path 1 is available for programmer use as needed.

424 load boots both chips using a fast serial boot stream, leaving the IDE operational as in the above environment.

1.1.3 Accessing Target Nodes from Host Chip

polyFORTH code may exchange data with the I/O support nodes on the Target chip using the Snorkel and Mark 2 Ganglia. The initial path for such transactions goes North one from Node 307, West 7 to Node 400, and South 1 which brings us to node 10400 in the Target chip using the transparent bridge. From there the path depends on which node is to be accessed.

The diagram below shows the Port Bridge being dynamically accessed via Snorkel and Ganglion (the dotted line) to perform operations on the Target Chip:



GA144 Host comm for Target / ADV7611

A similar procedure is used when booting from flash.

2. Resonant Devices at Low Frequency or with Low Q.

Our hypothesis was that we could connect a resonant device between a single GPIO pin and ground, then excite the device under program control and afterward maintain it in stable oscillation by pumping energy into the device synchronously, using it as a time base for applications that demand one whose frequency does not depend on Process, Voltage or Temperature as do our asynchronous circuits' internal operations. Our goals were to obtain this capability without adding any special circuitry within or outside the chip, without using more than one GPIO pin, and without any external components other than the resonant device itself. This was only an hypothesis because we could not find anything in the literature to suggest anyone else had actually tried such a thing; most crystal oscillator designs are dedicated, largely analog circuits, and most require two I/O pins.

2.1 32.~~758~~ 768 kHz Watch Crystal

In this exercise, we use a Seiko VT200F-12.5PF20PPM crystal which resonates at 32.768 kHz, with tolerance of ±20 parts per million (available down to five) and load capacitance of 12.5 pF (available down to four.) The package is a metal cylinder 6mm long and 2mm in diameter.

Using a GA144-1.10 (proto chip #004) in socketed test board #3 with V_{DD} at 1.80V, we connect this crystal between pin 16 (signal 300.17) and V_{SS}; as initially tested there are about 1.25" of wire (30 nH) between the crystal and V_{SS}, and about 2.25" of wire and PCB trace (57 nH) between the crystal and the chip pin.

Our motivation for using one pin rather than two, and for omitting other parts, is to prove that we can do without these things. Our view is that additional parts are only justified if they earn their keep, which includes any additional energy they may require by their mere presence.

2.1.1 Learning Start-up Requirements

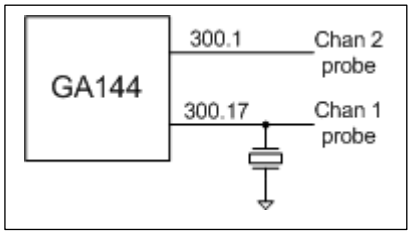
The first problem to be solved is to excite the crystal sufficiently for it to begin producing an oscillation whose amplitude crosses the input switching thresholds of F18A pads. Given the high Q of the crystal, this means we will need to excite the crystal very near its resonant frequency; we will have to do this using timing loops, and since the period of a timing loop depends on variables such as V_{DD}, temperature, and process variations, we know it will be necessary to search for this frequency. Until we experiment with a physical crystal, we won't know how narrow a frequency window we will have to hit, and we don't know how much excitation will be required.

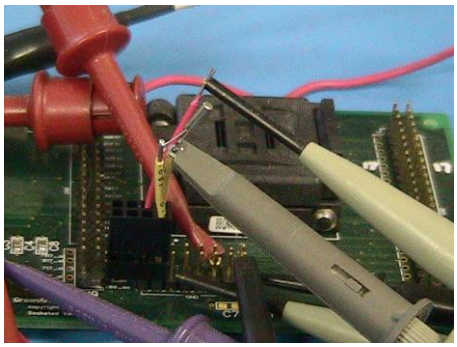
Initial investigation begins using our Interactive Development Environment (IDE) to program the chip and test the node under test. We also need to see what is going on at the pin, and we'll continue to need this until we are certain we can start the crystal successfully. For this purpose we connect a Tektronix P6131 10:1 passive probe (10 Megohms, 10.8 pF) to the "high" leg of the crystal about 1/4" from the can, and observe it using an analog input on an H-P 1631D logic analyzer (minimum sample interval of 5 ns.)

Our strategy will be to have the node under test stimulate the crystal with a given number of cycles of a square wave whose period is timed by a different number of iterations of an empty `unext` (micronext) loop. The code being used gives a resolution of one micronext iteration, or roughly 2.5 ns, for the period as a whole. When the desired number of square wave cycles has been completed, and the last phase driven was low / V_{SS}, the node will immediately set the pin to high impedance (250 megohms, 2.8 pF, plus, for now, the scope probe and wire/board parasitics.) We will watch the pin throughout the series of excitation cycles and will then look for a signal immediately following; the excitation square wave's period will be adjusted, manually, 2.5 ns at a step, until we see oscillation.

Since we may need to stimulate the crystal for a long time (many cycles) to impart the required amount of energy, we will use signal 300.1 as an output which goes high at the end of the stimulus phase when the crystal pin goes to high impedance; this may be used for scope triggering when the period of interest begins.

This diagram and phot reflect the initial test set-up:





The initial code for node 300 was this:

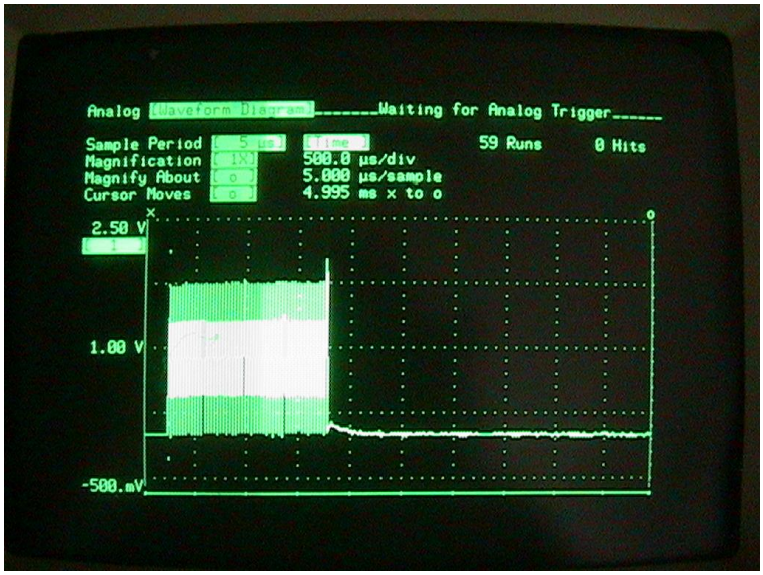
```
first phase exploring crystal excitation. br

light attempts to excite a crystal by giving i
t n+1 cycles of a square wave whose period is
k+2 unext iterations plus a little overhead. t
he square wave phases are half the period give
n with remainder if any added to the low phase
. after the final low phase, pin 17 is set to
high impedance and pin 1 is driven high to sig
nal beginning of the observation period. cr
use 3 vice 30003 and 2 vice 20002 to generate
pulses instead of square waves.
try1 makes an attempt to light the crystal usi
ng the period on the stack for the number of c
ycles given by literal in the definition.
try does this with a given period.
sweep makes a try1 using period on stack, shor
tening that period by 1 afterward to increase
frequency. push starting period onto stack ini
tially using nnn lit first in the ide. it is b
est to increase frequency since the chip slows
down as it heats.
```

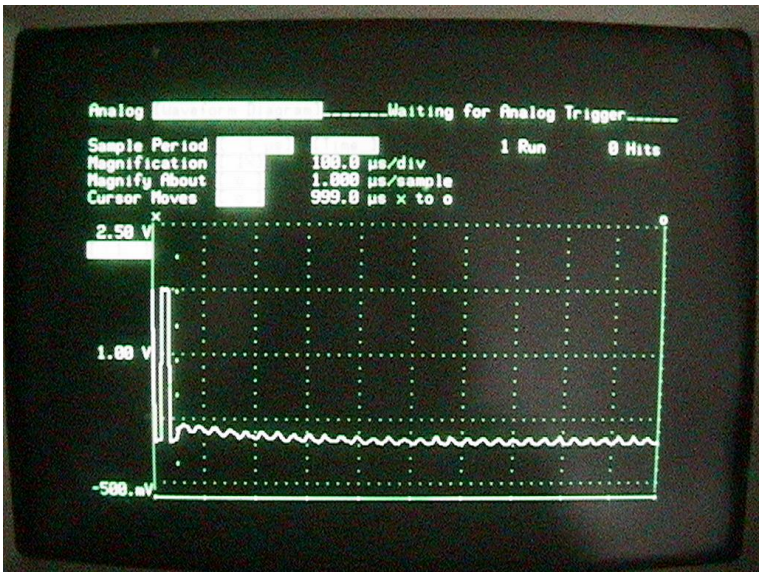
```
1332 list
32.768 khz xtal 300 node 0 org
light k n io b! for cr
2 30002 30002 .. !b !b dup 2/ for unext cr
2 20002 20002 .. !b !b dup 2/ over 1 and . + f
or unext next drop hi-z 3 !b ; cr
38 org
try 038 12138
try1 k-k 03A 50 200 light ;
sweep k-k' 03C dup try1 -1 + ; 03F
```

For the crystal frequency of 32.768 kHz, the period is 30.518 microseconds. At roughly 2.5 ns per `unext` iteration, the period would be roughly 12,207 loop iterations per cycle at typical room temperature. In reality the number should be a bit smaller than this because of the time required for the other work done during each stimulation cycle.

For each of the following tests we will manually try a range of frequencies, using **`sweep`** to search for the crystal's resonance. This range will be shifting as the temperatures of the chip and the room vary, so some patience will be required. Initially, we try 50 cycles of stimulus. The following trace shows no oscillation:

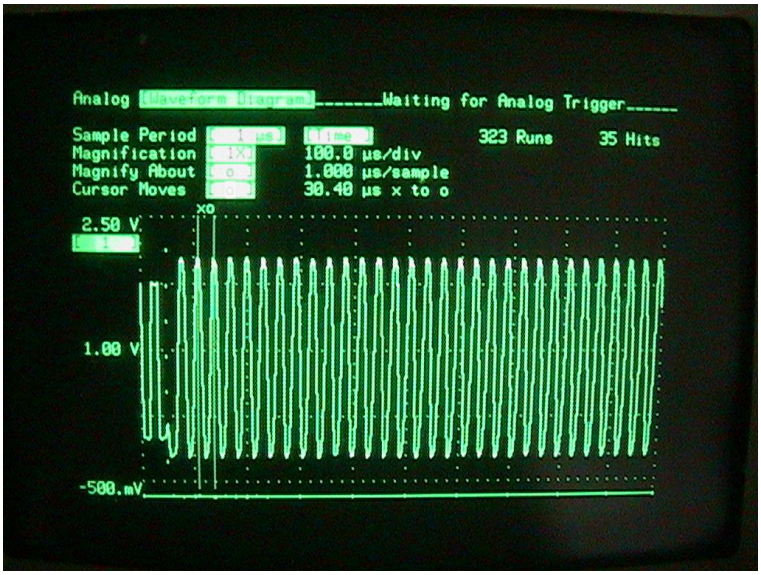


This is not what we need to see; even mechanically banging on the crystal should get it to oscillate at its resonant frequency with low amplitude. So we speculate that we simply have not put enough energy into it with this small sequence of square waves. Increasing the loop count in **`try1`** from 50 to 200, and setting the scope to trigger on the rising edge of 300.1, we obtain the following:



This is more like it! With more searching the resonance can be sensed, but the most we are getting out of the crystal is on the order of 100 mV peak to peak. Nevertheless, it is clearly resonating, given that we are seeing a signal at the correct frequency.

Increasing to 1000 cycles, we found the resonance; the maximum signal increased to +500, -125 mV and the resonance seems to be two or three of our period durations wide, which bodes well for our being able to automate this process! Now we go to 5000 cycles and are able to put enough energy into the crystal that it is exceeding our supply rails:



Interestingly, this resonance continues for quite a long time, even though the protection diodes are extracting energy each cycle. Further interactive experimentation followed, varying the start-up method, leading to these observations:

- We have to hit the right period ±1 count in order to hit the resonance for a crystal of even this low frequency but high Q, and the peak to peak voltage immediately drops by a factor of two or more when we are off this peak. Given that the period is on the order of 12000 counts, this means the useful resonance is on the order of 250 parts per million in period variation.
- Square waves put much more energy into the crystal than do pulses. For example, 5000 cycles of <10 ns spikes (V_{DD}, high-Z, V_{SS}, high-Z) only produce +1, -0.148V signal and the resonance is narrower. So square waves will certainly give us a faster and simpler start-up.
- Each increase in the number of driven cycles before releasing the crystal to resonate at high impedance increased the oscillation amplitude after a lengthy time delay. Over a range of 200 to 20,000 cycles this proved to be true, indicating that we are still adding energy to the piezoelectric device well after we have driven it hard enough to make our protection diodes start conducting.
    - After 5000 square waves, signal peak was 1.37V after 250 ms and 0.648V after 500 ms.
    - After 10,000 square waves, signal peak increased to 0.789V after 500 ms.
    - After 20,000 square waves, signal peak was 0.882V after 500 ms of free running oscillation at high impedance! That's a *long* time!
- These times are probably conservative because the scope probe in use is still loading the circuit down and doubtless extracting energy from the crystal.

Now that we know it will be feasible to start the crystal under program control regardless of the variables affecting our timing, we need to develop and test that program and then to implement the code for replenishing the energy of the crystal so it will oscillate indefinitely. Along the way we will be seeking ways to minimize the energy consumed in both starting and sustaining the oscillation.

2.2 10 MHz Ceramic Resonator for Ethernet

At about the same time as the above work was underway, Chuck Moore had attached a cheap 10 MHz ceramic resonator to the chip and had demonstrated he could start this low Q device with a considerably less painstaking effort than was required for the watch crystal. Later we adapted this code to give a time base for 10baseT transmit timing:

```
10 mhz ceramic resonator drive.,
each 50 ns edge sends a return instruction,
thru up port to tx pin node 317 which waits,
for the edge by simply calling the port.,
io power seems to be 70 to 90 uamps.,
,
node 317 must wait faithfully for every edge,
to keep this oscillator running reliably.,
,
note that after starting, the resonator decays
in about 10 us; so node 317 starts us when it
is ready to go by writing to the port.,
,
timing is actually quite tight in here; had to
fight to keep duty cycle lt 100 to prevent,
creep. also the resonator enthusiastically,
responds to pumping; rings have phase jitter,
so for this app with 20 mhz stimuli we must,
pump on every edge. ring code would be...,
ring 22 @ drop drop !b a push a! ! pop a! ..
```

```
728 list
417 tx osc 788 load exit 417 node 0 org,
,
drive 00 4n !b !b for unext drop ;
init 02 up a! @ drop .io b! left a!,
8 dup 0 20000 9 dup 800 30000,
...dup dup drop drop 9 for drive drive next,
15555 up 0 20000 .15555 up 800 30000,
...dup dup drop drop 22/1
go 22 . @ drop . !b !b a push a! ! pop a! go ;
26 reclaim exit
```

This code actually worked and was capable of timing 10baseT packet transmission, however was not practically usable because the resonators are not accurate enough in absolute frequency or thermal drift to meet the requirements of the receivers in typical 10baseT NICs. In fact we had to "pull" most of the resonators to produce intelligible bit streams.

2.3 Watch Crystal Mark 2 (AN012)

We decided to use the above watch crystal in the TI SensorTag work performed in early 2013 and documented in AN012. All that was required was to automate the start-up procedure and oscillation maintenance.

We did not need precise frequencies, particularly, but what we did need was a stable low frequency time base to measure the major polling intervals and to pace the I^2C bus clock without wasting energy on timing loops. The oscillator is placed in node 715 and the signal it generates is available at minimal power to nodes 709, 713 and 717 due to low-capacitance internal connections. These nodes provide their timing functions only when needed and are suspended when not. As a result, the average current used by the GA144 including its powering of the I^2C bus pull-ups between polls is very small... 14 to 15 µA of leakage, 28 or so µA to run the oscillator and another 2 µA for node 717 to count down for the next major cycle, for a total on the order of 45 µA or **95 µW average at 2.1V** to run the oscillator and count oscillator periods to produce lower frequency stimuli. We felt at the time that it was still possible to find ways to reduce the current used in driving the crystal. Here is the oscillator code:

```
32.768 khz watch crystal from 715.17 to gnd,
,
this code takes 27 ua at 2.1v using a seiko,
vt200f-12.5pf-20ppm crystal, 42 cents unit qty
from digi-key.,
,
-osc tries exciting the crystal with n cycles
of period k returning nonzero if it didn't,
come back high after last cycle.
clang searches for resonant frequency over a,
reasonable range. initially we use 5000 cycles
and may be able to shorten this. when we find
resonance, falls thru into prep which sets up
registers and finally we camp in run which is
the low power, low duty cycle oscillator.
try is test code for finding resonance.,
,
do not connect any kind of conventional probe
to the crystal; this oscillator will not work
if you load it down even that much.
```

```
980 list
715 xtal osc reclaim 10715 node 0 org,
-osc kn-f 00 io b! for,
..02 30000 !b dup .. 2/ dup for unext 06/1,
..20000 !b .. over 1 and .. + for unext next,
..dup or !b dup 30000 for,
....drop @b - -while next ;,
......then dup or pop drop ;
clang 14 14450 200 for dup 5000 -osc while,
..drop 1 . + next clang ; then pop drop ;,
prep 1F 0 20000 800 30800 0 20000 800 30800,
..dup up a! drop
run 2C !b !b @ drop run ;
try 2E dup 5000 -osc over 1 . + ;,
34 reclaim
```

This exercise clearly gave us what we sought: A time base for application activity that, **at 1.8V, would consume less than 81 µW average, including the mechanism to divide such an oscillator down to any desired lower stimulus frequency.**

3. High Frequency or High Q Oscillators

Our initial attempts at energizing a 10 MHz crystal using the above methods indicated that the frequency resolution possible with instruction loops was too coarse to excite the crystal within the narrow range prescribed by its Q except by extremely good luck. This notion was reinforced with our successful experiment in starting such a crystal with the above methods by using a 20 turn pot to make fine adjustments in V_{DD} and therefore in our instruction frequency. We were convinced by that experiment that we would not be running any 1-pin high frequency crystals unless we did something completely different.

3.1 Strategy

For several years we did not have any ideas about how to accomplish this without using a higher powered external oscillator (such as the Fox device we used in our initial 10baseT implementation) temporarily during excitation of the real crystal; no doubt this would have worked, however the cost would have been high in dollars per unit energy saved. The despair ended when Stefan Mauerhofer visited in the Spring of 2015, volunteering to test an hypothesis we'd discussed before.

3.1.1 Hypothesis

What, we asked, if a crystal could act as a mechanical filter to integrate a phase jittering stimulus whose average period was its resonant frequency? Perhaps we could achieve the effect of exciting the crystal at a frequency we could not achieve with a stable signal by giving it a sequence of cycles at two frequencies varying by our minimum resolution. Perhaps by using a Bresenham interpolation we could excite the crystal at effective frequencies whose resolution was considerably finer than we could achieve with straight timing loops.

3.1.2 Evidence of Validity

Stefan undertook to build a mechanism for doing this, using crystals between a GPIO pin and ground with no additional circuitry besides parasitics. To all of our surprise and delight, he was able to start oscillation in a wide range of high frequency crystals! By the time he had to return to Switzerland, we now knew this was possible, although there remained open questions; Stefan had procured a large set of crystals ranging from <1 MHz to >10 MHz and, for example, while most of them could be started with the code he was able to complete while in Cheyenne, some could not.

Thus, the question became one of understanding the process and controlling it well enough that this could be moved from an interesting laboratory demonstration to a reliable capability usable in production.

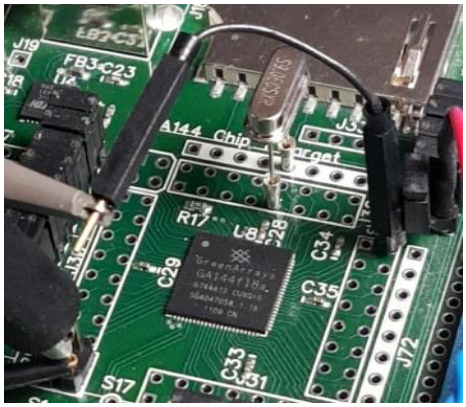
3.1.3 Motivations and Action

Our motivation was initially to stop depending on an external Fox oscillator for Ethernet transmit timing, a motivation that became stronger when a PCB assembly house closed its doors and stole our very expensive and hard-to-get reel of those oscillators. The final push came from a design project in which we could reduce the cost per instance of the full design by several thousand dollars if we could reliably run a crystal at for example 9 MHz. Therefore by the end of 2015 it was show time for this application capability.

Our first step was to take Stefan's code and test configuration, instrument it, and use this to study the variables while making improvements in the process and in the code based on what we learned in the study. The central purpose of the study was to yield a technique whereby we could reliably start the crystal in an automated way.

3.2 Instrumentation

To minimize parasitics, we made a clean physical test platform by soldering two machined pin receptacles from a DIP socket between pin 10715.17 and ground on the target chip of an EVB001. This pin was chosen because it is shared with three other nodes (10709, 10713 and 10717) along the top edge of the chip, thus allowing independent nodes to monitor the crystal while adding absolutely no parasitics to affect the loading on the resonant device. Then, to observe activity on that pin, nodes 10717, 10617 and 10517 are programmed to output onto 10517.17 the signal at the crystal as processed by the Schmitt triggers on the input of 10715.17, using the code shown below:




```
796 list
-- xtal osc,
mon 10717 +node 10717 /ram down /b 0 /p,
..10617 +node 10517 /ram down /a up /b 0 /p,
..10517 +node 10517 /ram up /a io /b 0 /p,
```

```
10717 monitors phantom pin from 10715 and,
..sends an io value thru 10617 to 10517 for,
..setting 10517.17 to track the crystal pin.,
,
note that failing to initialize return stack,
..caused us to crash and apparently fail to,
..oscillate on 2 jan phase jitter still ???
```

```
810 list
717-517 monitor,
reclaim 10717 node 0 org,
mpre 00 30000 up 15555 io 20000 up 15D55 io,
..dup dup drop drop down b!
mon 0C a! ! a! dup ! !b mon ;,
,
0E reclaim 10517 node 0 org
zz -1 dup push dup push dup push dup push,
......dup push dup push dup push dup push push
mon begin begin @ !b unext unext mon ;,
07 reclaim exit,
```

3.2.1 Mechanism

The only data we get without being invasive are histograms of high impedance rings after each excitation as a function of effective (average) excitation period. We invented a multi-node delay line to store such data on-chip:

```
this code implements a data delay line in an,
..arbitrary number of nodes, with 64 words of,
..delay for each node.,
,
delay line nodes contain no code; all ram is,
..used as a fifo. stack is init circularly as,
..shown so that t is value of the first instr,
..sent by shove and s is the second instr.,
,
node 414 serves as control via port execution;
shove pushes a word into the delay line,,
..returning the oldest element.,
feed in @p prime w/focus if needed to store,
..only and ignore data pushed out of the line.
slurp is called by a ganglion transaction.,
..send n-1 count of words to push out of the,
..fifo and return in reply message.
```

```
800 list
414 et al delay line,
13 10414 node 16 org reclaim
shove n-n 10 right a! ..,
..12 @p ! .. !b @ !b @p ..,
..14 up a! @ drop @ push right a! ..,
..19 ! @p ! .. !+ !b ..,
..1B up a! @ drop pop ;
prime n 1E shove drop ; 20 0 org
slurp 00 pop b!,
..@b for 0 shove !b next ;,
,
06 reclaim exit,
,
initialize delay line nodes as follows...,
+dly ion +node /b /p FBB2 9E27,
..over over over over over over over,
..10 /stack 0 /a ;,
right left 10415 +dly
```

With this tool, we can run a sequence of trials to excite the crystal and record its response to each trial for later recovery and display using **`slurp`** .

3.2.2 Measures

polyFORTH code in the host chip initiates trials of different periods and, when complete, retrieves the data from the delay line into an array from which various analytics are recovered, including data across multiple runs:

- Minimum and maximum number of "hits", or trials producing one or more cycles of ringing, in a set of runs.
- Mean number of "hits" in a set of runs.
- Span of hits, the number of trials in the range from the first to the last hit across multiple runs.
- Lowest and highest periods of hits across multiple runs.
- Minimum and maximum number of hits in multiple runs.
- Maximum number of non-hits within the range of periods producing hits across multiple runs.

3.3 Study of Variables

There are many variables that appear, or seem likely to, affect the reliability of crystal start-up. Therefore we undertook to explore each variable systematically, using the above instrumentation. The variables in question are as follow, along with such answers as the data suggest:

- Length of energizing sequence.
- Does applying Bresenham to half cycles rather than full cycles help or hinder? *Appears to hinder, at least in the first set of cases checked.*
- Does one energize contaminate the next? *Yes, for example we get false "hits" if we begin a new trial while the crystal is still ringing energetically from the preceding trial.*
- In light of that does the sequence of trial frequencies matter? *We learned that the sequence does matter, but not neceassarily for this reason.*
- Does series resistance help or hinder? *We did not actually study this variable.*
- Does parallel capacitance help or hinder? *We did not actually study this variable.*
- Does weak 40k pull-down of the pin help in any way? *It appeared to.*
- Do we have an algorithm that is guaranteed to start a crystal in one trial? *No, but manageable.*

3.3.1 Baseline Data

The baseline conditions are: Excitation at effective periods of 9.0 thru 15.63 with 10,000 half-cycles (5,000 full) at each period, taking a maximum of 50 cycles of ringing initially. Because of the way in which Stefan's code worked, 1.1 was the first period tried and the last was 16.0. Tek P6131 probe on high side of crystal. The initial set of crystals were at 9.0, 10.0, 12.0, 15.0 and 18.0 MHz. The range, and length of delay line, were extended before including the 12, 15 and 18 crystals.

3.3.2 Experimentation

There ensued an exhausting, ten-day effort at the beginning of January 2016 during which all conceivable variables were explored, in the quest for a one-pass start-up procedure that could be guaranteed successful. As it turned out, we concluded that this was a futile quest in that there was a nonzero probability of failure to start the crystal in a single sweep of the excitation frequency range, regardless of variation of the parameters. Some things appeared to help, but none led to a definitive one-pass solution, so we opted for an adaptive procedure instead. In the process several things were learned; for example, that it was important to sweep stimulus frequency from low to high rather than from high to low, because it was possible to induce spurious oscillation modes at higher frequencies than true crystal resonance; for example, our 9.0 MHz crystal could be excited, and induced to oscillate stably, at 9.5 MHz! So many intriguing mysteries, so little time.

3.4 Code Clean-up for Production

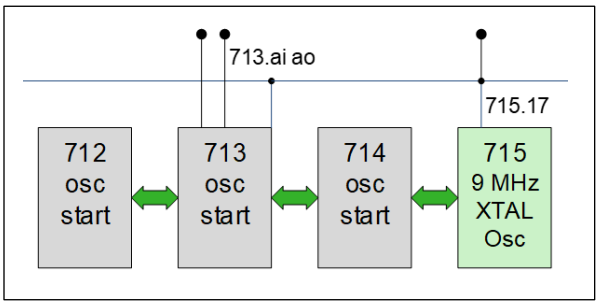
After abandoning the quest for the Grail of a deterministic single-pass frequency sweep for successful excitation, we made two instances of the crystal oscillator to meet the needs of the design project underway at the time: A 9 MHz oscillator for serial communications with intolerant receivers, and a 10 MHz oscillator for 10baseT Ethernet.

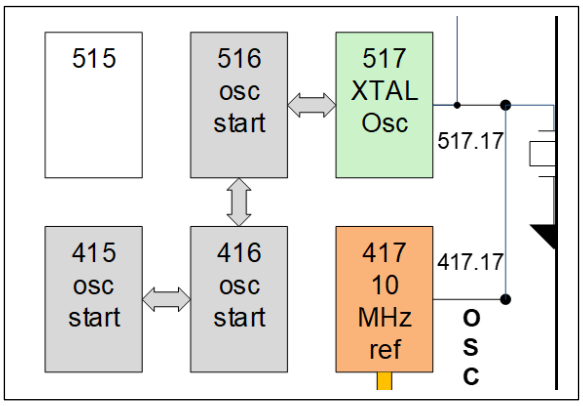
In the process we simplified Stefan's code, which had been written openly to facilitate experimentation, into a more compact, final form as a cluster of four nodes. Three of these are used only during the start-up procedure, while the fourth is used to excite the crystal and, once oscillation has started, to keep it oscillating continuously as we do the watch crystal earlier in this paper.

We had intended to add code to the start-up sequence generating nodes so that they could monitor for oscillator failure and re-start if necessary. We did not actually do this for three reasons. Firstly, unless these nodes have a shared pin on which to observe the oscillator, the act of monitoring via a COM port presents failure modes of its own. Secondly, we have not seen an oscillator stop, once started, without resetting the chip or injecting noise onto the pin, usually through a probe and cable. When the crystal is soldered to a board a short distance from the chip, the path for such large noise is nonexistent. Finally, in the design effort that motivated this work, the PCBs were required to have watchdog timers anyway; so by coupling the process of watchdog tickling to oscillator operation, we already had a way to handle that problem if it ever arose.

3.4.1 Floorplan

Here are two examples of crystal oscillators from a practical application.





The oscillator in node 715 runs at 9 MHz and distributes its output using the shared pin seen by nodes 709, 713 and 717. In this case, it would be feasible for node 713 to monitor the oscillator and potentially restart it if necessary. The oscillator in 517 is used to provide transmit timing for the 10baseT Ethernet NIC, but its start-up nodes are landlocked and could not monitor the oscillator without depending on multiport writes, requiring very careful design. In each case the crystal is simply connected between the GPIO pin of the respective XTAL Osc node and ground.

3.4.2 Oscillator Code

In each of the floorplans above, there is basically a chain of four nodes terminating at the pin connected to the crystal. The last node has code to make a trial to excite the crystal as commanded and report the results, normally switching to oscillator maintenance if the excitation appears to have worked. The first node controls the start-up procedure, sequencing through excitation periods in units of 1/64 of a unext execution time. The second node generates sequences of square wave periods to implement the desired fractional period, buffering them in the third node's RAM, after which it commands the fourth node through a trial and reports results. The code discussed below is designed to run the 10 MHz crystal for Ethernet transmission, using pin 517.17. Code for another frequency and/or another pin differs only in the floorplan and therefore register settings for different COM ports or pins.

3.4.2.1 Oscillator Node

This node performs operations at the command of the sequence generator until the crystal has responded to excitation, after which it assumes its dedicated role of running the crystal.

```
this node excites the crystal, detects ring,,
..and syncs to run oscillation as commanded.,
,
a param node, monitor node or pin,
b points to io,
,
-osc pumps energy into crystal using sequence,
..of half-cycle periods fed from param node.,
..then senses rise indicating ring within time
..out period on order of 4 ms. true if no ring
,
fire called externally to attempt start-up,
..with pattern chosen in param node. returns,
..0 for failure, 1 for ring but not started,,
..-1 for ring and start. select no start by,
..un-commenting the 'continue...' line.
```

```
792 list
517 xtal oscillator node,
reclaim 10715 517 node 0 org
-osc -t 00 right a!,
..20000 30000 04 7 for over unext 07/2,
..@ for 08 !b @ push begin unext next,
sense 0A wpd 15555 !b,
..30000 for 0E @b - .. -while next ; 12,
....then pop dup or ;,
,
valz 13 0 20000 800 30800 ;,
,
fire 19 -osc dup down right a! if dup or ! ;,
..continue... then 1 ! ; exit,
..go... then -1 ! valz valz up left a!
sync 24 3FFFF 0 !b @ drop for 28,
..!b !b @ drop next @b 2000 and if ;,
then drop sync ;,
2B reclaim exit
```

P is initially set to **right** (node 516) from which we expect to be commanded **B** is set to **IO** initially. The commands expected are calls to **fire** which makes an attempt to start the crystal, reports results as shown, and falls through into **sync** to maintain oscillation if the start-up appears to have succeeded.

fire simply invokes **-osc** which does the hard work of exciting the crystal and seeing if it rings; if **-osc** returns true, the crystal is not ringing so we return zero to the start-up nodes (via **A** which is already set in that direction). The commented second line of this definition is only used when studying; normally, when **-osc** says the crystal is ringing, we return -1 to the start-up nodes, set up the stack using **valz** and fall thru into **sync** which maintains the oscillator.

sync has a loop which may be used to check the port from the start-up nodes at infrequent intervals, using the commented code, so that the start-up nodes could stop the oscillator and later restart it. These modes are not used in operational code but may be useful in an application that only needs to run the crystal part of the time.

-osc sets **A** to point at the start-up nodes and fills the stack with values for **IO** of x20000 (driving pin 17 low) and x30000 (driving it high), leaving the high state on top. It then receives a sequence of data from the start-up nodes: One word with number (-1) of half-cycles (edges) to give the crystal, followed by that many delay values (-1) to use after each edge. The first edge presented to the pin is rising, followed by the first delay value, and so on. We expect an even number of edges so that the last edge drove the pin low. After the excitation sequence of jittery square waves has been presented to the crystal, we set the pin to weak pull-down (this has empirically helped) and wait for a rising edge on the pin. If one is seen, we return 0 meaning the crystal has responded, and fall through; if it is not seen before about 4 ms have elapsed, we return a nonzero value meaning that the crystal has not responded.

sync then, normally, is an infinite loop maintaining the oscillation of the crystal. Initially we set **IO** to zero, so that the pin is at high impedance and so that wakeup direction is 0 (wake on pin high) whereupon we read the pin to wait until it is in fact high. We then begin a sequence of pulsing the pin high for approximately 5 ns, returning it to high impedance and waiting for the pin to go low. We pulse the pin low for approximately 5 ns, returning it to high impedance while we wait for the pin to go high again. As noted above, commented code in **sync** could provide a way to interrupt the oscillation assuming it is running (if it is not, the loop would never finish.)

3.4.2.2 Sequence Generator

This node is commanded by the start-up control node via port execution. It generates a pattern in the RAM of the next node in sequence and finally commands the oscillator node, using port execution in both cases. Nothing other than the pattern is stored into the RAM of the node in between. The node is initialized with **P** pointing at the port from the start-up control node and **B** pointing to the port leading to the RAM node. Within the RAM node, **P** points to us and **B** points to the oscillator node.

```
                                                794 list
p control node; b is ram node,                  - 416 oscillator sequence gen,
,                                               reclaim 10713 416 node 0 org,
/rewind set a in data node to 0,                /rewind 00 @p !b ; .. dup or a! ..
/push push value on data node return stack,     /push n 02 @p !b !b ; .. @p push ..
/read fill data node memory,                    /read 04 @p !b ; .. begin @p !+ unext ..
!osc writes value to xtal node's port,          !osc n 06 @p !b !b ; .. @p !b ..
/write send data to xtal node,                  /write 08 @p !b ; .. begin @+ !b unext ..
@osc reads value from xtal node's port,         @osc -n 0A @p !b @b ; .. @b !p ..,
,                                               bres so-sov 0C over . + -if a ;,
bres returns value for next half cycle. during  ..then 0F -64 . + a 1 . + ; o init -33 s pos
..interpolation s is positive /64 fraction,,    'step sub val rep -r 14 push /rewind,
..t is ramp var, a is the integer value.,       ..15 a! 63 dup push /push /read,
'step given /64 sub, base val, count rep of,    ....19 -33 begin bres !b next 1D,
..halfcycles-1, loads pattern into ram node,,   ..1D @p !osc .. fire ..,
..calls fire in xtal node, feeds it pattern of  ..1F pop dup !osc /push /write @osc ;,
..val and val+1, returns result from xtal.      24 reclaim exit,
```

'step is the function called in this node by start-up control after pushing three parameters onto the stack. Its purpose is to run a trial at a particular excitation period in the oscillator node, reporting results.

sub is the low order six bits, and **val** the high order bits, of an excitation period in units of 1/64 of a `unext` instruction time. **rep** is the count of halfcycles-1 to run. This must be an odd number so that the number of halfcycles actually run is even; the **fire** function in the oscillator node assumes that the last driving signal in the excitation attempt was low, and if an odd number of halfcycles are run the test will always (falsely) conclude that the crystal is ringing.

The first thing **'step** does is to store a sequence of halfcycle durations into the RAM of the next node, using the following functions that send port-execution to the RAM node; those starting with slash operate on table in its RAM:

/rewind stores zero into **A** in the RAM node as the memory pointer for stores and fetches.
/push pushes the number given onto the return stack in the RAM node.
/read is followed by a sequence of cell values that are stored into RAM node memory starting at **A** and of whatever length the loop count on its return stack dictates (normally 63 for 64 values.)
/write sends RAM data to the oscillator node starting at **A** for the count on return stack. Assuming there are 64 values in RAM, the length of the sequence transmitted may be any even value up to 262144.
!osc sends the number given through the port into the oscillator node.
@osc reads result from oscillator node and delivers it to us via the port.

We save the count of halfcycles for excitation on the return stack, set up the origin (0) and length (64) of the sequence in the RAM node, and push loop count for generating 64 values to our own return stack. The high order (integer) part of the excitation period **val** is saved in **A**. Two values are placed on the stack: The fractional part **sub** of the desired period, and the ramp variable **o** initially -33 (based on 32 as half of fraction range). We then call **bres** 64 times, storing the resulting values into the RAM node.

bres does a stepwise Bresenham interpolation. The fractional part is added to the ramp variable; if the variable remains negative, we return the base integer interval from **a**. If it is no longer negative, we subtract the range 64 from the ramp variable and return the interval from **a** made longer by one `unext`.

After the RAM is filled, we command the oscillator node to make a **fire** attempt, send it the number of excitation half-cycles **rep** which we'd saved on the return stack, and command the RAM node to send that number of periods. Finally we retrieve the result from the oscillator node and return it to the start-up control.

3.4.2.3 Start-up Control

The final node in the start-up effort controls the making of excitation trials and recognition of success. This version continuously attempts crystal excitation, sweeping the selected frequency range repeatedly until success is observed. One side effect of doing it this way is that, even if a crystal is not present, a square wave signal will be appearing at the oscillator node's pin continuously. If this is not wanted, comment the call to **retry** at the end of the loop in **start**.

```
b is oscillator sequence generator,
,
trial tries excitation at the period n and,
..returns a neg value if the osc node decided,
..to try running with it. n is in unext cycles
..with 6-bit fractional part.,
,
start begins with a period much longer than is
..likely for the xtal in use, trying shorter,
..periods in 1/64 unext or roughly 42ps mean,
..i.e. interpolated period. returns to await,
..when the osc node commits itself.,
```

```
796 list
- 415 osc start + take data,
reclaim 10712 415 node 0 org
strt 00 leap await ;,
,
trial n-r 02 @p !b .. @p @p @p ..,
..dup 63 and !b 2/ 2/ 2/ 2/ 2/ !b,
....rep 9999 !b,
..@p !b .. 'step .. @p !b @b ; .. !p ..
start*retry 0E then initial 1024,
..959 for -1 . +,
....dup trial - -while drop next retry ;,
..then pop drop ;,
,
18 reclaim exit,
```

strt is the entry point which calls **start** and returns to warm multiport execute when **start** returns.

trial makes one trial of the given half-cycle period in 1/64 unext units. The period is divided into integer and fractional parts, which are passed as arguments to **'step** in the sequence generator node along with the parameter 9999 indicating that 10,000 half-cycles of excitation should be made. The value returned is the result code generated by the oscillator node (nonzero if successful excitation).

start runs a sequence starting at a period of 1024 for 960 trials, shortening the period by 1/64 `unext` each step. Thus the half-cycle periods actually tried in this case run from 1023 to 64 or 16 to 1 `unext` for full periods of 32 to 2 `unext` times (76.8 to 4.8 ns) however there are about 18.5 ns of overhead per half-cycle in the **-osc** code, so we actually have periods ranging from about 113.8 to 41.8 ns. These are estimated frequencies of roughly 8.79 MHz to 23.3 MHz. The numbers defining the range may need adjustment for other crystals or other operating conditions (PVT).

With the above code, we jump to **retry** if a sweep fails, as noted above. Otherwise, when the oscillator has been started, we return to **strt** and this node returns to its warm address waiting for instructions from any COM port.

Stefan has contributed two observations about the production code above:

1. The code in **trial** assumes that the integer part of the half-cycle excitation period will not need to be larger than 11 bits, for 2048 `unext` times or about 5 µs; this gives a full cycle of 10 µs or 100 kHz square-wave frequency. This can be extended to be on the order of 50 kHz by masking to 17 bits after the final `2/` in **trial**. To use this algorithm with lower excitation frequencies it would be most effective to reduce the fractional part to 5 or 4 bits and change **bres** accordingly.

2. While the start-up control node is left in its warm multiport execute using **await** the other two start-up nodes are left executing the ports through which they are commanded during start-up. By adding more port execution functions these nodes could also be left in warm conditions.

Here is an example of load descriptors for the above oscillator cluster:

```
naked 10mhz xtal exit 0 io right 517 +nd,
..ram 0 right down 516 +nd,
..pat 0 down left 416 +nd,
..strt 0 left 0 415 +nd
```

4. Conclusions

As a result of this work, we are comfortable with designs that include quartz crystals at frequencies up through 10 MHz. To specify a higher frequency crystal, more testing and evaluation should be done. Additionally, there are many things about this work which we do not understand well enough to explain confidently, such as how it is that the crystal can be excited successfully with the frequency jittery signal in the first place, and why the spurious oscillation at a higher frequency is possible. We have not done an exhaustive literature search but if in fact no one else has written about these phenomena they might be good topics for a high school science project by some interested student.

4.1 Implementation

After starting an oscillator there will be a single node engaged in tracking the crystal and keeping it running. How the signal is distributed depends on application requirements and also on frequency; at 10 MHz and below it's feasible to pass events at double the crystal frequency (20 Mhz event rate and below) while at higher frequencies there is eventually no time to write through a port and so the oscillator signal will have to be monitored by a shared or phantom pin, or by a separate GPIO pin, out of necessity rather than merely as a convenience.

During oscillator start-up, the crystal node will have a single port that's used to control this process. That may be any of its ports except a port that will be sending timing signals in the application. If the start-up mechanism remains in place after the process has succeeded, it could be used to re-start the crystal should it ever fail. In addition it is conceivable that this could provide an avenue whereby the crystal may be stopped under program control and restarted later.

4.2 Energy consumption

The average power required to simply run a crystal is mainly a linear function of its frequency, and it is practically all core power in executing instructions at whatever duty cycle its frequency demands. I/O power used is immeasurable with test equipment limited to 1 µA resolution. If anything is done with the crystal signal on chip, this implies one or more additional nodes running at the same duty cycle and will increase average power proportionally. Here are some data points for just the node that keeps the crystal running:

Freq	Current mA	Avg Pwr mW
32.768 kHz	0.023	0.041
9.0 MHz	1.131	2.036
10.0 MHz	1.255	2.239
15.0 MHz	1.883	3.389

In addition, there is a very small current used on the appropriate I/O bus. For node 10715 this is $V_{DD}A$ and what we see there is 1 µA leakage, 43 µA when running the 10 MHz crystal.

These numbers compare favorably with the Fox F210 series 1.8V oscillators such as the one we used on some of our Ethernet interfaces. The Fox devices are rated for an IDD of 2.5 mA for 9 and 10 MHz, 3.5 mA for 15 MHz, thus using more power than do our software oscillators above. When the price of the oscillator (over $2.00 in quantity 1000) is considered, the comparison is *very* favorable.

In contrast, minimal F18 timing loops burn about 3.2 mW continuously. The watch crystal is a couple orders of magnitude cheaper than this, with of course less resolution. However even at 9 MHz it will be a challenge to measure a time delay while also running the crystal. So high frequency crystals are generally only justifiable if an accurate or stable time base is required; however if the watch crystal gives sufficient resolution it is clearly the best choice for low power time measurement regardless of accuracy or stability requirements.

4.3 On External R-C Networks

For a purely capacitive load, the energy required in Joules to change its state rail to rail is

$$J = \frac{{V_{DD}}^2\, C}{2}$$

When using an external (series) R-C network to measure time, the energy actually required depends on how it is going to be used. When driving the network from one rail to the other, its state will be seen to change by pin wakeup when it crosses approximately $V_{DD}/2$ which occurs at about 0.7 TC. Thus, if the circuit is sitting at one rail or the other, it should take >0.7RC seconds to cross the switching point V_{IT+} and trigger pin wake-up.

Assuming that we are driving the circuit in saturation, which is effectively true for the first 0.7TC from a rail, the drive transistors will look like 20Ω resistors. The energy required to take an RC network to this point depends on the ratio of R and C!

Ignoring time for the moment, the charge required to reach a given voltage on the capacitor, in Coulombs, is equivalent to the integrated current over that period of time. The RC step response is basically linear over the first 50% of the voltage swing so we may envision this as a constant current phenomenon for most of that time. The bottom line of that is that we may treat average current as being proportional to C. Bearing in mind that $P=I^2R$ this suggests that our choice of R and C to make a given time constant has considerable impact on the power dissipated in the resistance while charging or discharging the cap. Increase R by a factor of 10 and decrease C (and therefore I) by a factor of 10, the time constant is unchanged but P is now one tenth what it was due to the I^2 .

While this is important to understand, its practical utility is limited by the ranges of available parts values. Let us look at some examples. A microfarad is a good sized capacitor but using a 1 MΩ resistor. which is also good sized, we get a 1 second time constant (or about 0.7 seconds to 0.9V from either rail). The above observations suggest that to achieve shorter time constants than one second we should reduce the size of the cap and leave the resistor alone. So let's look at the 1s time constant (0.7s delay) in more detail.

To charge 1µF to 0.9V requires 900nC. The average current required to achieve this in 0.7s is 900e-9C / 0.7s or 1.29µA. Power is then I^2R or $(1.29e\text{-}6)^2$ * 1e6 or 1.66 µW. To charge 1 nF to 0.9V requires 900pC. The average current required do to this in .7 of its 1ms time constant is 900e-12 / 0.7ms or 1.29µA once again, giving the same average power as above.

These numbers are theoretical and have not been verified empirically, nor have we designed an efficient strategy for running such time delays repetitively (which appears to require expending time and energy to move the network back to a rail, with no obviously efficient way to time that unless we are running two RCs alternating), but for some applications this may be worth breadboarding and exploring. Another strategy Daniel Kalny suggests is to simply run the capacitor's voltage between VIT+ and VIT- for a smaller interval without some of the problems. Another interesting aspect of this problem is that if there is enough inductance in the RC path one must delay between turning on drive transistor and checking the pin; we have observed, and triggered on, inductive kick-back with relatively short wires.

4.4 Expensive Active Probe for Free

In the course of this work, we needed active scope probes that minimally loaded the circuits under test. When all that is needed is a digital probe, one of our GPIO pins works very well as the "Probe tip" (2.5 pF, 250 MΩ). The signal this probe detects may be routed and analyzed on-chip, or routed out another GPIO pin, without imposing any further load on the circuit under test. While we could theoretically use an analog node in a similar fashion, the frequency range would be much lower and the signal would need work to linearize and calibrate. For our purposes in studies like this, the active logic analyzer probe we get by employing one of our GPIO pins is very good and is far cheaper than a passive scope probe of equal quality, if there actually even exists one.

5. Revision History

REVISION	DESCRIPTION
100910	First Draft in HTML
171102	Update to App Note format and inclusion of recent work, pre-publication draft.
171106	Release for Publication

GreenArrays®

Application Note AN002 *Revised 11/06/17*

IMPORTANT NOTICE

GreenArrays Incorporated (GAI) reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to GAI's terms and conditions of sale supplied at the time of order acknowledgment.

GAI disclaims any express or implied warranty relating to the sale and/or use of GAI products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

GAI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using GAI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

GAI does not warrant or represent that any license, either express or implied, is granted under any GAI patent right, copyright, mask work right, or other GAI intellectual property right relating to any combination, machine, or process in which GAI products or services are used. Information published by GAI regarding third-party products or services does not constitute a license from GAI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from GAI under the patents or other intellectual property of GAI.

Reproduction of GAI information in GAI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. GAI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of GAI products or services with statements different from or beyond the parameters stated by GAI for that product or service voids all express and any implied warranties for the associated GAI product or service and is an unfair and deceptive business practice. GAI is not responsible or liable for any such statements.

GAI products are not authorized for use in safety-critical applications (such as life support) where a failure of the GAI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of GAI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by GAI. Further, Buyers must fully indemnify GAI and its representatives against any damages arising out of the use of GAI products in such safety-critical applications.

GAI products are neither designed nor intended for use in military/aerospace applications or environments unless the GAI products are specifically designated by GAI as military-grade or "enhanced plastic." Only products designated by GAI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of GAI products which GAI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

GAI products are neither designed nor intended for use in automotive applications or environments unless the specific GAI products are designated by GAI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, GAI will not be responsible for any failure to meet such requirements.

For current information on GreenArrays products and application solutions, see www.GreenArrayChips.com

Mailing Address: GreenArrays, Inc., 821 East 17th Street, Cheyenne, Wyoming 82001
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com