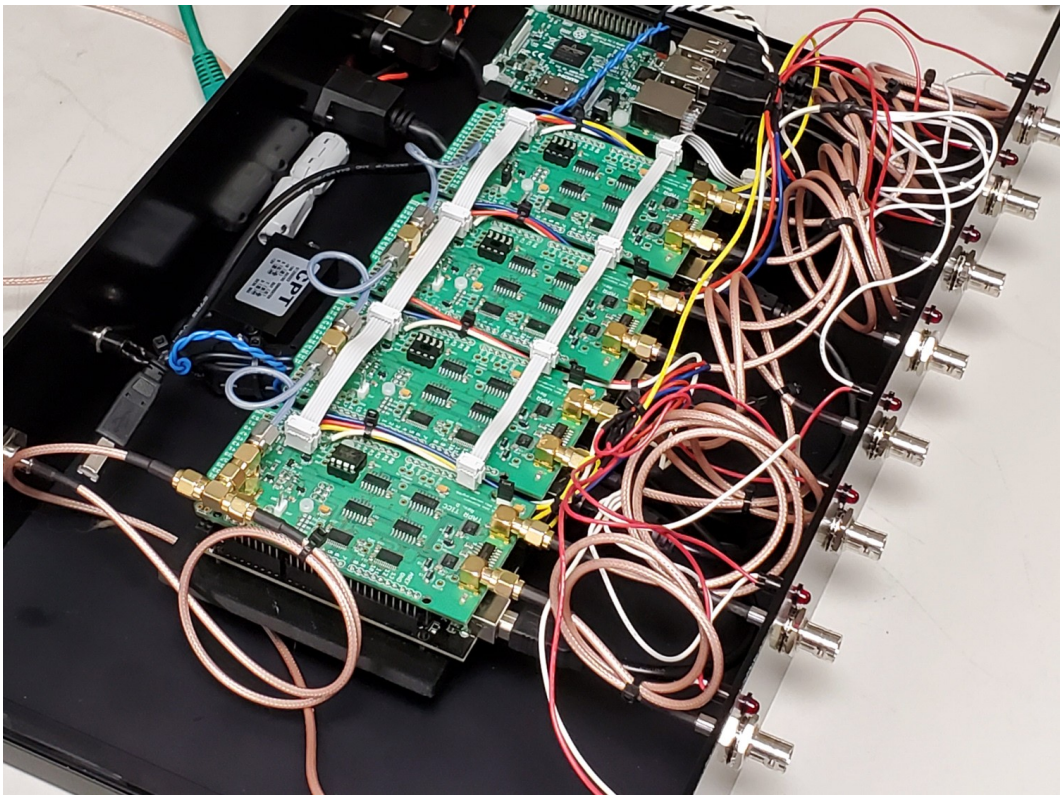
# multi-TICC:
# A Multi-Channel Timestamping Counter based on the TAPR TICC

by John Ackermann N8UR



## Introduction

The TAPR TICC (https://tapr.org/?product=tapr-ticc) is a two-channel timestamping counter with a resolution of <60 picoseconds. It is a “shield” that mounts on an Arduino Mega 2560 processor board, and it communicates via the Arduino’s USB port.

A timestamping counter is one that maintains an internal timescale, for example a counter clocked by an external 10 MHz reference that starts at power-on and runs continuously. The timescale allows tagging, or “stamping” the time that external events occur in relation to the timescale. Every time a pulse appears on one of the TICC’s two input channels (called “chA” and “chB”), the TICC outputs the timestamp of that event and the channel on which it occurred.

A timestamp is a low-level measurement that can be used directly to compare the period and stability of the input signal compared to the reference clock. It can also be used to compare timestamps from channel to channel. For example, if separate pulse-per-second signals are applied to chA and chB, you can calculate the time interval between them simply by subtracting the timestamps. Or, subtracting the the previous timestamp from the current one allows you to determine the period of the signal on one channel. Timestamps are very useful building blocks.[1]

While the TICC is a two-channel device, it provides on-board headers that expose clock and synchronization signals. By connecting these signals across two or more TICCs, and providing a common 10 MHz reference input, multiple TICCs can share a single timescale. In this way counters with four, six, eight, or even more inputs can be assembled, and you can compare measurements across all channels.

If a multi-TICC system is coupled with a single-board computer such as a Raspberry Pi, it can serve as a “timestamp appliance”: users can make a network connection to the computer and download live data from any or all of the channels. Input signals can be connected and disconnected at any time without impacting data logging from other channels. When a channel comes “on line” its timestamps will be based on the same timescale as the other channels.

This means that the multi-TICC can be kept running across many measurement cycles, and measurements from multiple channels and multiple measurement cycles can be matched in sequence by their timestamps.[2] With the current TICC firmware, the timescale will not roll over for about 4 years after system power-on, Typically, only the loss of the 10 MHz reference signal will require a reset of the multi-TICC system.

This application note describes the main aspects of assembling and using a multi-TICC: hardware setup, firmware updates, and host processor software; and also provides some preliminary test results.

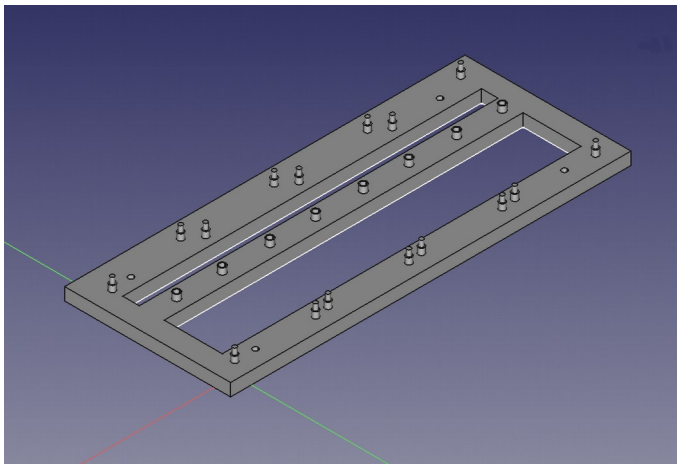
1 With the current TICC firmware, the timescale will not “wrap” (overflow and restart at zero) for about 5.8 million years.

2 Note, however that as timestamp values increase over time, precision in floating point conversion can be lost. See Appendix A for more details.

## Hardware Configuration

In the multi-TICC, one board is configured as the “master” and provides clock and synchronization signals to the other “slave” boards. All boards run off a common 10 MHz reference clock. Their USB outputs are sent to a host processor which makes the data streams available via ethernet.

To make it much easier to hold the boards together, I created (with a lot of help from Mike, W8RKO) a 3-D printed carrier that will align and secure 4 boards. That carrier can in turn be mounted to a base plate.



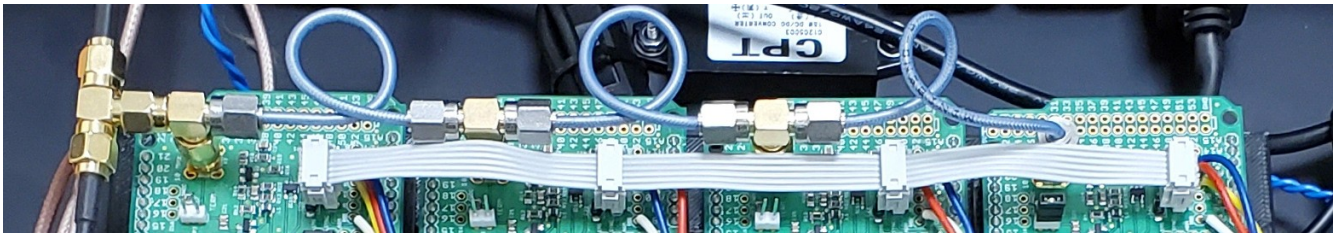
The carrier design files (and all files referenced in this document) are available from the TICC github repository (https://github.com/TAPR/TICC).

There are three points of electrical interconnection between the boards in a multi-TICC system.

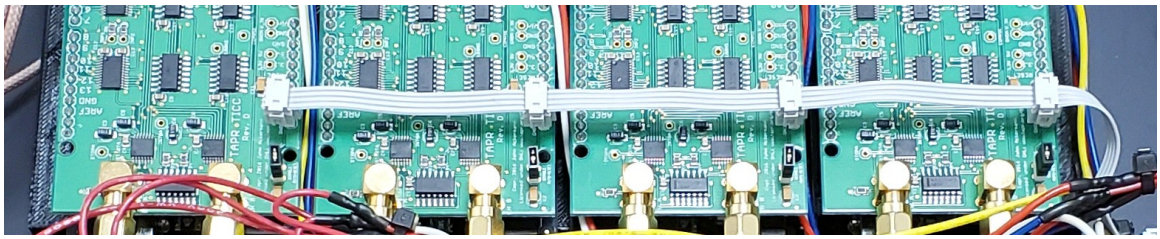
1. A common 10 MHz clock is applied to all boards through either a passive splitter or a distribution amplifier like the TADD-1. The “TERM” termination jumper should be installed on all boards. Note that if you use a passive splitter, you may need to increase the 10 MHz drive level to compensate for the ~7 dB loss in a typical 4-way splitter.

**IMPORTANT NOTE:** I originally used daisy-chained connectors for the 10 MHz clock signal, but that seemed to result in odd behavior, with boards showing differing ADEV floors; in the worst case, ADEV on one board would rise above $1x10^{-10}$ at one second, or one board would show occasional phase jumps of several hundred picoseconds. The mechanism for this is as yet unknown, but testing has shown that feeding the boards separately is a better approach.

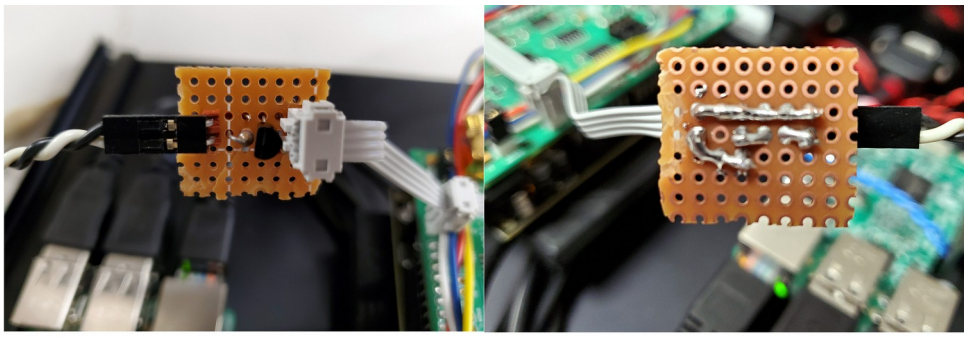
2. As shown below (but ignore the daisy-chained coax lines!), the three pins of the JP2 header are connected in a daisy-chain fashion across all the boards. The easiest way to make this cable is to use 0.1 inch spaced IDC ribbon cable connectors with 6 positions – just make sure you plug the same row of the connector onto each JP2!. If you use the 4-board frame described below, the nominal spacing between JP2 headers is 2.25 inches.

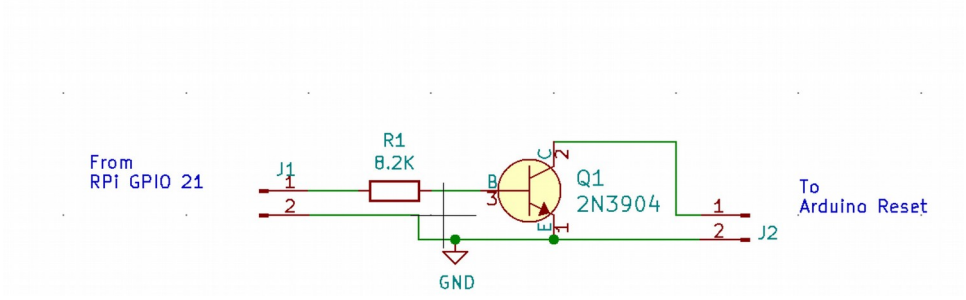


3. As shown below, the RESET lines of all the boards are also connected in parallel, so that they can be reset simultaneously. To do that, solder a 2 pin header in the holes marked “GNDB” and “RESET” located on the right front of each board, and made a daisy-chain connector tying them together. Again, it’s easiest to do this with a 4 pin ribbon cable. One note: with the reset lines in parallel, if one of the TICCs is powered down, it will pull the reset line of the others low with the result that they will not start. Make sure all the TICCs are powered up when the reset daisychain is connected.



4. If using a Raspberry Pi as the host computer one of the Pi’s GPIO pins can be used to cause the Arduinos to reset via software command. The problem is that the Arduino uses 5V logic and the Raspberry Pi is 3.3V. The very simple level translator shown below can be used to interface between the two boards. It’s easy to assemble this on a small piece of perf board as shown below.





5. Install shorting blocks on the “DISABLE AUTO-RESET” header (JP1) on each board. If this is not done, each time a connection is made to the Arduino serial port, that TICC will reset and that will cause synchronization problems. With this jumper installed, you will not be able to upload new firmware without performing tricks. The multi-ticc_updater.py program discussed below does the appropriate tricks and you can use it to update firmware even if the DISABLE AUTO-RESET header is shorted. If you don’t use that tool, remember to remove the jumper block before programming!

6. On “slave” boards, remove IC10 (the 12F675 PIC). This is because the coarse clock signal generated by the PIC on the master board is routed via JP2 to the slave boards.

With these changes, your set of boards is ready to emerge as a multi-TICC!

## Firmware Configuration

The firmware shipped on recent TICC units (version 20170309.1) has basic multi-TICC capability included. A newer version (20191202.1, available in the github.com:/TAPR/TICC repository) adds three helpful features for multi-TICC use:

First, you can assign channel IDs other than A and B.

Second, you can use a “PROP_DELAY” configuration variable to set an offset value in picoseconds for each channel which will be added to the values reported from that channel. This allows you to compensate for the length of the interconnect cables and other sources of board-to-board delay. (PROP_DELAY serves the same purpose as the existing “FUDGE0” variable, but I’ve been requested to make two delay settings available. The PROP_DELAY and FUDGE0 settings are additive.)

Third, a bug fix enables an external LED to show that the board is being clocked. If the LED is attached to the last board in the string, it will give you some assurance that the board interconnections are correct. The LED can be attached to the A11 pin on the right rear area of the board. (External channel activity LEDs can be connected to the A12 and A13 pins as well.)

To enable multi-TICC operation, use the configuration [Y] command to set the master board (typically the first in the chain) as [M]aster, and to set the other boards to [S]lave mode. Optionally, use the [N] command to change the channel ID to any single printable ASCII character. Since the master board by default has channels A and B, it’s sensible to set the second board to “C D”, the third to “E F” and so on. Finally, if desired use the PROP_DELAY or FUDGE0 variables to set the offset in picoseconds. Note that the Arduino serial monitor program is a bit funky when it comes to data input; you’ll probably have better luck if you use a “real” serial terminal program to set configuration parameters.

## Data Interface and Host Processor

If you have 4 TICCs configured configured in master/slave mode, you also have 4 USB ports carrying serial data. It would be nice to consolidate those data streams, and maybe even make them available via Ethernet. A Raspberry Pi is an excellent tool to provide this function – its four USB ports are a perfect match for a four-board multi-TICC configuration. I've written a simple Python TCP server program to serve as a data aggregator spitting all the TICC data out over a telnet connection.

In order to keep this document reasonably short, and also because the code is still being revised, I won't go into a lot of details, but here are the basics on the main program as well as some other programs that make managing the system easier. At some point when things have had some time to be debugged, I will create a ready-to-run SD card image so all you'll need to do is plug that in.

Described in a text file located with the programs is a description of software prerequisites that need to be installed to use the software, as well as some other setup information. It describes how to persistently assign serial port names dev/ttyTICC0, dev/ttyTICC1, dev/dev/ttyTICC2, and dev/ttyTICC3 to the four devices. Those names are used in all the programs described here, and the assumption below is that the system contains four TICC units. All the programs are installed in the /home/pi/ directory on the Raspberry Pi.

`multi-ticc_server.py` – if run with no options provided will look for four TICC units on connected to the Raspberry Pi serial ports. It outputs several streams of data on different TCP ports, by default:

port 9190: Outputs multiplexed data from all active TICC channels, unsorted
port 9191: Outputs multiplexed data from all active TICC channels, sorted by timestamp
port 9192: Outputs data from chA
port 9193: Outputs data from chB
port 9194: Outputs data from chC
port 9195: Outputs data from chD
port 9196: Outputs data from chE
port 9197: Outputs data from chF
port 9198: Outputs data from chG
port 9199: Outputs data from chH

There are many tools you can use to connect to the server from another machine and access the data. The easiest is with the telnet command:

```
telnet server.ip.address 9190
```

To make it easier to log the data to a file, you can use the Linux netcat program:

```
nc server.ip.address 9192 > datafile.dat
```

`multi-ticc_server.py` can be left running on a console and will show startup and connection status. Unfortunately, at this point the program can only support one client connection per data stream, but I'm hoping to add multi-connection support.
Here are some other utility programs that make managing the multi-TICC easier:

`multi-ticc_reset.py` will momentarily send Raspberry Pi GPIO pin 21 high, which if the level translator circuit described above is installed, will cause the TICCs to reset. Run this after starting the Raspberry Pi and before attempting to connect to the TICCs; on startup the TICCs normally freeze and kicking them with this program will cause a clean restart with all boards synchronized. NOTE: multi-ticc_server.py will perform a reset when it starts, so you should not ordinarily need to use this program.

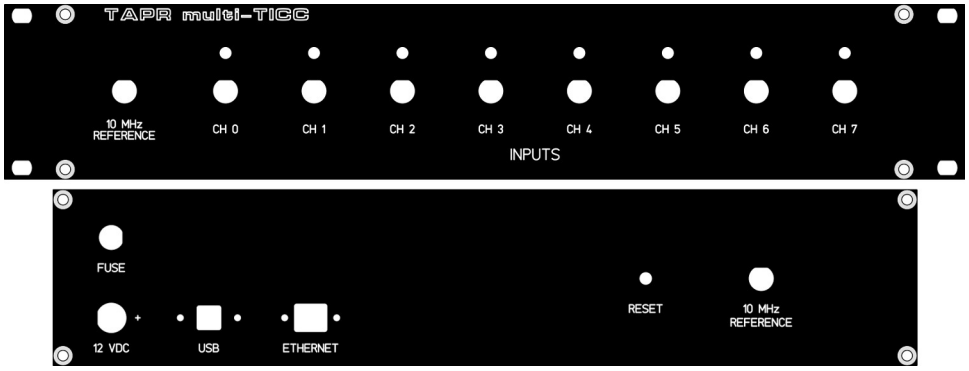
`multi-ticc_updater.py` will update all TICCs with a new firmware file in hex format. Supply the file path and name as a command-line argument. The multi-ticc github file includes the support files requires (`ticc_avrdude.conf` and the firmware `.hex` file; make sure these are installed in the /home/pi directory).

`miniterm.sh` is a shell script that will launch a simple terminal program allowing you to communicate with one TICC. Just give the TICC serial port name as a command line argument.

`multi-ticc_server.service` is a file that can be placed in `/lib/systemd/system/` to cause the server to automatically start at runtime, and also to restart if it fails. After putting it in the directory, run `sudo sysctl enable multi-ticc_server`.

## Miscellaneous

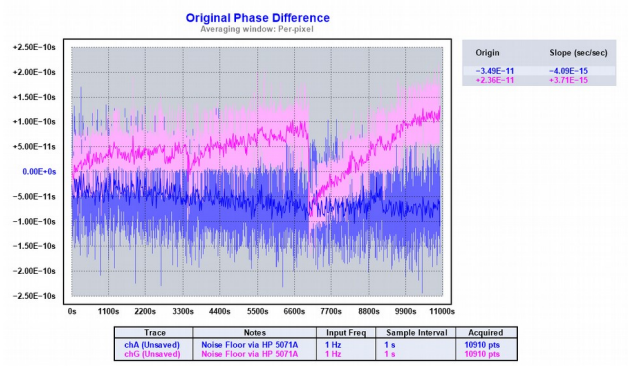
If you want to make your multi-TICC all pretty, I've designed a complete 2U rack enclosure using the Front Panel Express design tools. The files are available at github.com/TAPR/TICC/multi-ticc/



# multi-TICC Performance

Performance tests on the multi-TICC indicate that very little if any performance is lost compared to a stand-alone TICC. However, those tests have also shown that when you are trying to match 4 devices and 8 ports to picosecond levels, there are a lot of factors that make life very interesting.

The main thing learned during testing is that using a simple daisy-chain for the 10 MHz clock feed to the boards results in strange behavior. Either one of the boards shows somewhat higher or lower jitter, resulting in inconsistent ADEV, or alternatively phase glitches can be introduced on one or more channels. Here are two examples from early testing.



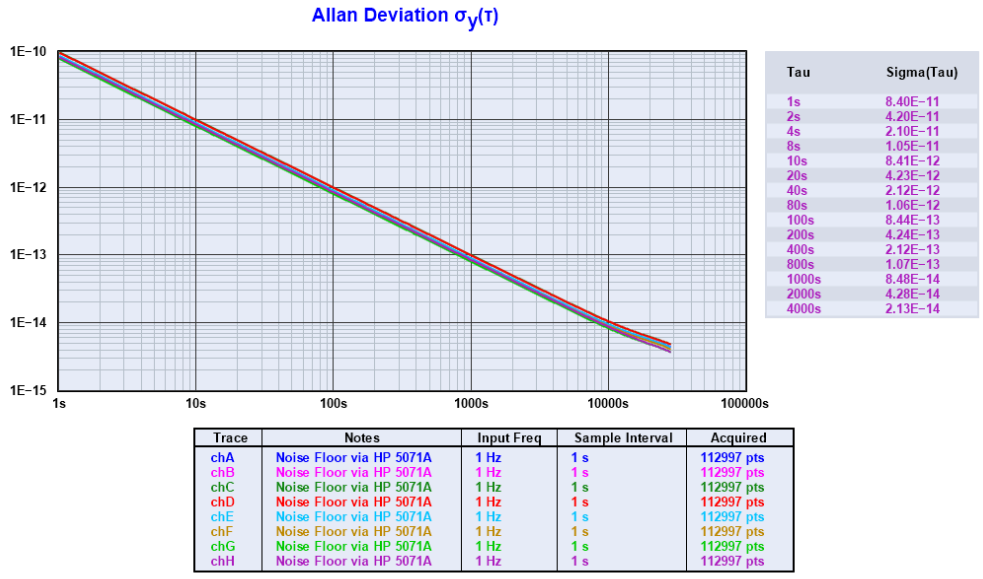


Based on those results, I am driving each TICC with one of the outputs from a passive 4-port splitter. The jumper on each TICC is installed to provide 50 ohm termination to the 10 MHz signal.

All the results shown below were taken with this clock configuration. I suspect further testing to determine optimal 10 MHz input signal levels will be worthwhile and might produce slightly better jitter results.

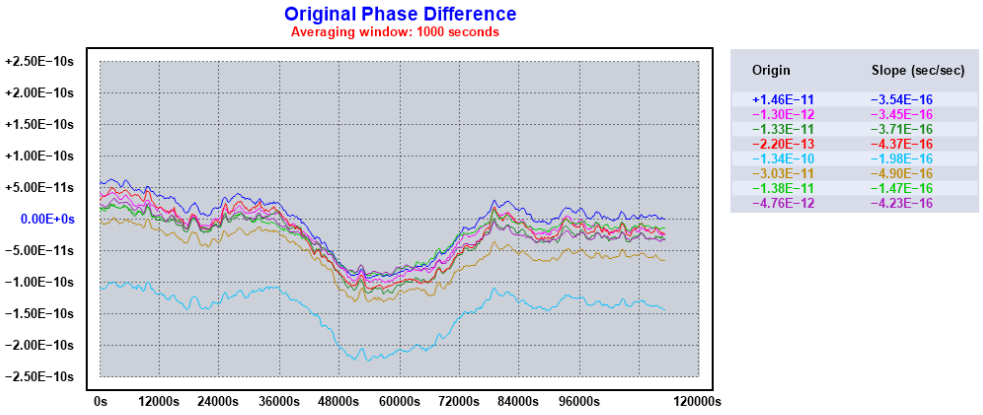
The final test configuration used an HP 5071A Cesium standard with high-performance tube to provide both 10 MHz and 1 PPS signals. The 10 MHz signals were routed as described above, with a 3dB attenuator at the input to the TADD-1 to reduce the input to about 10 dBm. The PPS signal was fed into a 3dB attenuator and then to a daisy-chain of cables using BNC tee connectors. At the end of the chain is a 50 ohm termination. The jumpers between channels each have about 2.4 nanoseconds delay.

With all that said, here are results from a 125,000 plus sample test of the multi-TICC, starting with plots showing the ADEV of each channel and the phase record (ignore the apparent offset of channel E in the phase record; that's a plotting anomaly).




| Tau | Sigma(Tau) |
|---|---|
| 1s | 8.40E−11 |
| 2s | 4.20E−11 |
| 4s | 2.10E−11 |
| 8s | 1.05E−11 |
| 10s | 8.41E−12 |
| 20s | 4.23E−12 |
| 40s | 2.12E−12 |
| 80s | 1.06E−12 |
| 100s | 8.44E−13 |
| 200s | 4.24E−13 |
| 400s | 2.12E−13 |
| 800s | 1.07E−13 |
| 1000s | 8.48E−14 |
| 2000s | 4.28E−14 |
| 4000s | 2.13E−14 |

| Trace | Notes | Input Freq | Sample Interval | Acquired |
|---|---|---|---|---|
| chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chB | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chC | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chD | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chE | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chF | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chG | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chH | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |

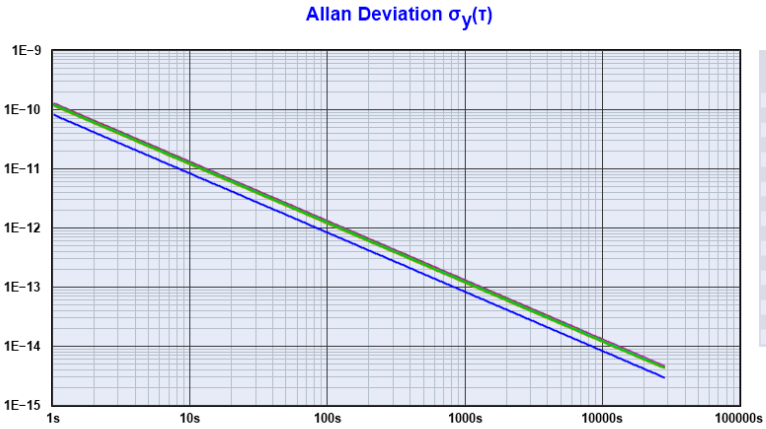



| Origin | Slope (sec/sec) |
|---|---|
| +1.46E−11 | −3.54E−16 |
| −1.30E−12 | −3.45E−16 |
| −1.33E−11 | −3.71E−16 |
| −2.20E−13 | −4.37E−16 |
| −1.34E−10 | −1.98E−16 |
| −3.03E−11 | −4.90E−16 |
| −1.38E−11 | −1.47E−16 |
| −4.76E−12 | −4.23E−16 |

| Trace | Notes | Input Freq | Sample Interval | Acquired |
|---|---|---|---|---|
| chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chB | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chC | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chD | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chE | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chF | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chG | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chH | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |

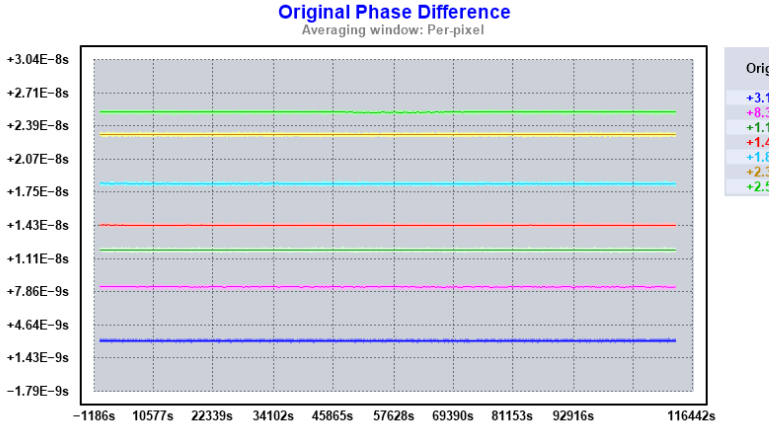
The ADEV readings are tightly grouped at just under $1x10^{-10}$, slightly worse than typical single-TICC performance around $8x10^{-11}$. Note the dip in the middle of the phase chart. My best guess is that this was caused by temperature changes in the lab overnight. Though it looks significant here, remember that the change is only about 100 picoseconds!

In some situations the multi-TICC is used in time interval rather than timestamp mode. In this case, channel A will typically be the START input and channels B through H the STOP inputs. The time interval is (chX – chA). The following plots show the result of that operation.




| Tau | Sigma(Tau) |
|---|---|
| 1s | 1.22E−10 |
| 2s | 6.10E−11 |
| 4s | 3.06E−11 |
| 8s | 1.52E−11 |
| 10s | 1.22E−11 |
| 20s | 6.15E−12 |
| 40s | 3.06E−12 |
| 80s | 1.53E−12 |
| 100s | 1.22E−12 |
| 200s | 6.12E−13 |
| 400s | 3.06E−13 |
| 800s | 1.53E−13 |
| 1000s | 1.22E−13 |
| 2000s | 6.14E−14 |
| 4000s | 3.05E−14 |
| 8000s | 1.52E−14 |
| 10000s | 1.22E−14 |

| Trace | Notes | Input Freq | Sample Interval | Acquired |
|---|---|---|---|---|
| chB - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chC - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chD - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chE - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chF - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chG - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chH - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |




| Origin | Slope (sec/sec) |
|---|---|
| +3.11E−9 | −4.27E−17 |
| +8.32E−9 | −8.38E−17 |
| +1.19E−8 | +3.83E−17 |
| +1.43E−8 | −1.07E−16 |
| +1.83E−8 | −4.51E−17 |
| +2.31E−8 | −1.20E−16 |
| +2.53E−8 | −7.55E−17 |

| Trace | Notes | Input Freq | Sample Interval | Acquired |
|---|---|---|---|---|
| chB - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chC - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chD - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chE - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chF - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chG - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |
| chH - chA | Noise Floor via HP 5071A | 1 Hz | 1 s | 112997 pts |

You can see that the phase change in the middle of the run is no longer visible. That is because it was common mode to each channel (i.e. a change in the phase of the HP 5071A 10 MHz vs. PPS outputs that affected both the measurement channel and channel A equally, so subtracting channel A removes the phase change). You can also see the increasing phase delay of each channel as the cable length (and delay) between it and channel A grows.

Using Tom Van Baak's command line ADEV and statistics tools, I extracted the following measurements from the run:

| | Picoseconds | | | | | |
|---|---|---|---|---|---|---|
| | ADEV @ 1 SEC | MEAN | SDEV | MIN | MAX | RANGE |
| **chA** | 8.38e-11 | 14904 | 65.2 | 14882 | 1492 | 462 |
| | | 5 | 2 | 7 | 89 | |
| **chB** | 8.64e-11 | 15215 | 64.5 | 15189 | 1524 | 546 |
| | | 1 | 0 | 2 | 38 | |
| **chC** | 1.00e-10 | 15735 | 71.7 | 15705 | 1576 | 602 |
| | | 7 | 5 | 6 | 58 | |
| **chD** | 9.72e-11 | 16095 | 72.7 | 16067 | 1612 | 574 |
| | | 6 | 6 | 0 | 44 | |
| **chE** | 9.03e-11 | 16336 | 65.2 | 16312 | 1636 | 503 |
| | | 0 | 2 | 1 | 24 | |
| **chF** | 8.56e-11 | 16738 | 63.0 | 16707 | 1676 | 560 |
| | | 6 | 5 | 2 | 32 | |
| **chG** | 7.89e-11 | 17214 | 54.6 | 17191 | 1723 | 464 |
| | | 7 | 5 | 7 | 81 | |
| **chH** | 8.35e-11 | 17430 | 57.6 | 17404 | 1745 | 483 |
| | | 7 | 2 | 2 | 25 | |

I attempted to work out the channel-to-channel delay from these measurements:

| | Mean Phase Difference (picoseconds) | | | |
|---|---|---|---|---|
| | Raw Values | | Minus Cable Delay | |
| | chA | Prev CH | chA | Prev CH |
| **chB** | 3106 | 3106 | 706 | 706 |
| **chC** | 8312 | 5206 | 3502 | 2806 |
| **chD** | 11911 | 4599 | 4711 | 2199 |
| **chE** | 14315 | 2404 | 4715 | 4 |
| **chF** | 18341 | 4026 | 6341 | 1626 |
| **chG** | 23102 | 4761 | 8702 | 2361 |
| **chH** | 25262 | 2160 | 8462 | -240 |

N = 112,970

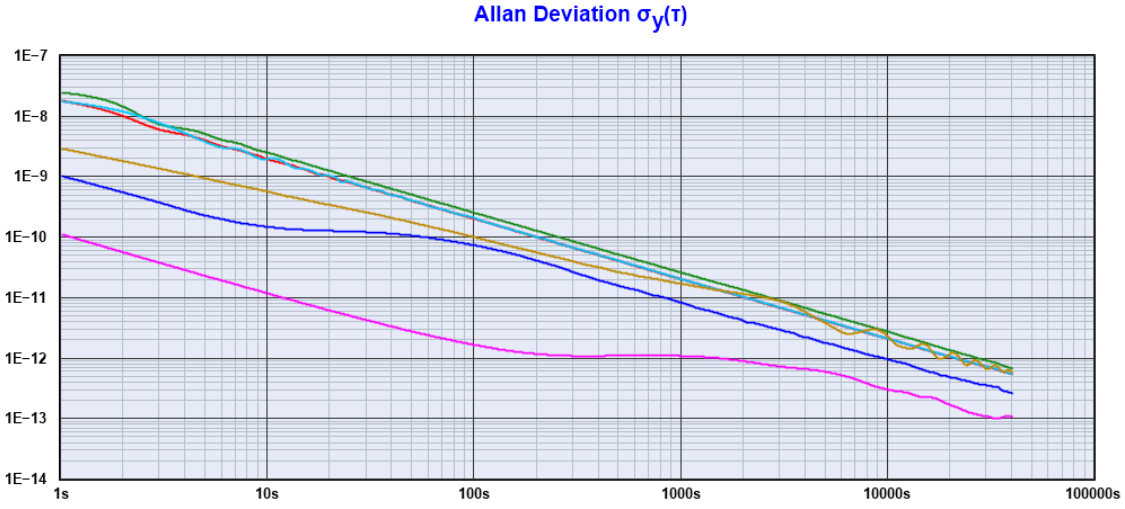
Calculations from TVB ADEV1 and STAT

In the table above, the "chA" columns show the subtraction of channel A from each succeeding channel. The "Prev CH" columns show the subtraction of the prior channel (e.g., the "chC" row shows (chC – chB). The "Raw Values" are simple subtraction of the mean values reported above. The "Minus Cable Delay" values subtract the delay of each of the jumper cables between inputs, which were 50 cm pieces of RG-316D with a measured delay

of about 2.4 nanoseconds. In an ideal world, the `Prev Ch Minus Cable Delay` values should be the same for each channel. It's easy to see that's not the case here.

One would expect the two channels on each board (e.g., chA and chB) to have only a small delay because both are fed from the same clock signals within the board. The delay between boards would be greater because there is an increasing delay in the propagation of the 10 MHZ and 10 kHz clock signals from board to board. A third consideration is how close to simultaneously the boards boot up. Finally, the phase of the 16.67 MHz CPU clock (which is free-running) may affect the initial starting condition. More experiments may determine what causes the range of delays seen here, and how to minimize the range.

To finish things up, here is an example of "real world" data capture from a multi-TICC with the PPS from six GPS units connected to channels A through F:




| Trace | Notes | Input Freq | Sample Interval | Acquired |
|---|---|---|---|---|
| CNS Clock II | via HP 5071A | 1 Hz | 1 s | 160294 pts |
| Z3805A | via HP 5071A | 1 Hz | 1 s | 160294 pts |
| Z12-REF | via HP 5071A | 1 Hz | 1 s | 160294 pts |
| NetRS1 (no ext ref, clock steering on) | via HP 5071A | 1 Hz | 1 s | 160294 pts |
| NetRS2 (no ext ref, clock steering off) | via HP 5071A | 1 Hz | 1 s | 160294 pts |
| NetRS3 (ext ref, clock steering off) | via HP 5071A | 1 Hz | 1 s | 160294 pts |

# Appendix A – Big Timestamps and Floating Point Accuracy

The TICC uses an unsigned 64 bit integer to count the number of 100 microsecond "ticks" since the system was started. That count acts as the system timescale. 64 bits is a very big number, and at 100 us per tick it will take over 5 million years for the TICC timescale to overflow and wrap around to zero. That's very cool – or is it?

The TICC does all its calculations with integers and the results retain the same accuracy as the value of the timestamp grows larger (*i.e.*, as the "seconds" part grows from 0 to some very large number). A timestamp with seconds in the billions still retains the ~50 picosecond resolution of the TICC results. But the external software used to analyze TICC output files will almost certainly convert the data to floating point format in order to perform the required math functions, and that's where a problem arises.

Floating point numbers are simply numbers with both an integer and a decimal part, like "1234.56789". The problem is that base-10 numbers with fractional parts cannot always be represented precisely in binary form. Converting a base-10 number to binary and then back again may result in a surprising difference between the before and after values.

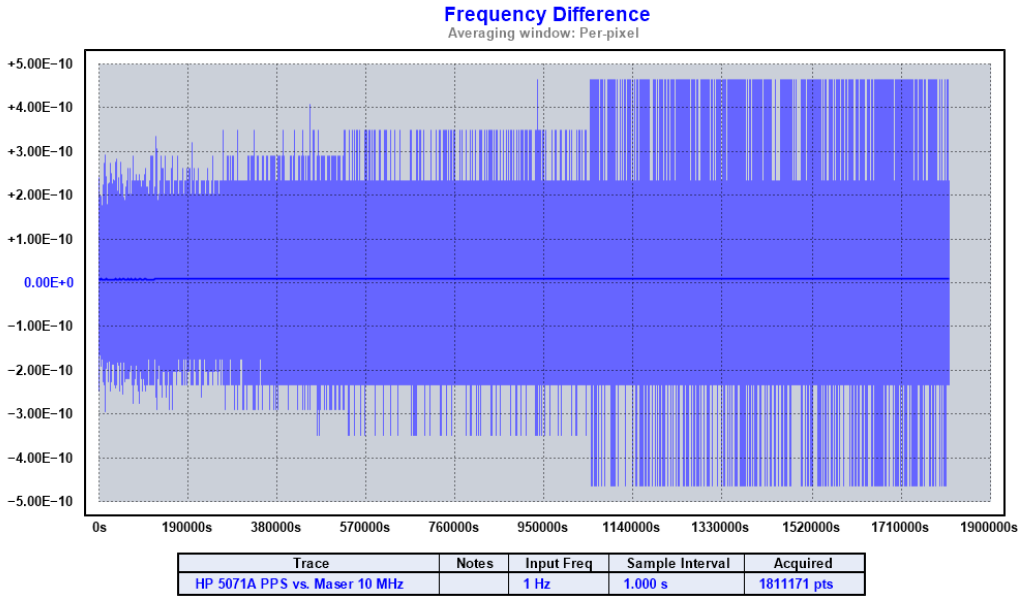
Most computers and computer languages use a standardized set of rules to handle floating point numbers. Under those rules, a floating point number can have just under 16 decimal digits of precision. Any number of the 16 digits can be to the left or to the right of the decimal point, but the total can't exceed that value.

If the number is less than 1, you can represent it accurately to 16 decimal places: 0.1234567890123456. But if it begins with 1,000,000 you only get 9 accurate decimal places, because 7 of the 16 digits are to the left of the decimal point: 1234567.890123456. There may be additional decimal digits in the result, but they won't be meaningful.

To reiterate, this limitation of accuracy is inherent in the computers we are using. Although there are ways to process data with fractional components using integer math, they are inconvenient, slow, and not very practical in a complex program. So this isn't the "fault" of the application software; it's just the way computer programs work.

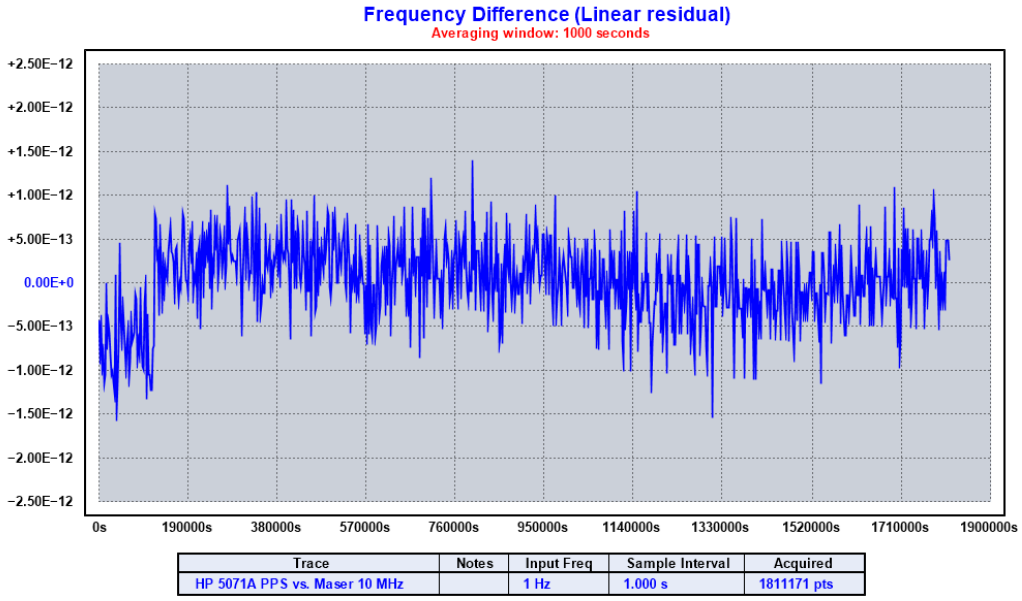
Since the TICC outputs data with 1 picosecond precision (12 decimal places), that means timestamps with an integer part greater than 9,999 will have a theoretical loss of accuracy when converted to floating point as only 11 of the 12 decimal places will be valid. Things aren't quite that bad, though, because the actual TICC resolution is about 50 picoseconds with the last 1 ½ digits of the result being noise (i.e., you could round the TICC results to the nearest 50 picoseconds and not lose any meaningful information). As a result, the earliest that the effect might be noticed is when the timestamp reaches about 50,000 seconds.

Here is a TimeLab frequency plot of a very long (1.8 million second) TICC timestamp capture:




You can see that there are stair-step increases of noise during the course of the run. This is the result of integer-to-floating-point conversion and lost precision (and thus increased apparent noise) as the timestamp values get greater. If you look at the data closely, you'll find that these stair steps occur first at about 55,000 seconds, and again at about 100,000 seconds, just as the discussion above would predict.

Here's the same plot with an averaging factor of 1000 applied:



Now the quantized step-increases are no longer present. I'm not sure of the exact mechanism that TimeLab applies when it does averaging of timestamp data, but reducing the number of samples brings the data back into a range where floating point accuracy isn't degraded.

Ways to pre-process TICC data to reduce the effects of floating point accuracy loss include:

1. Reducing the integer range of the timestamps (*e.g.*, if the integer value of the first timestamp in the data series is 100000, subtract 100000 from that and all subsequent values).

2. Convert the timestamps into phase difference records by subtracting from each data point the value of the prior point.

3. Sometimes, where the DUT and the reference are drifting only very slowly with respect to each other, and the fractional part of the data is not near 0 or 1, you can chop off the seconds part of the data completely and treat the decimal part as if it were a phase difference value.

**In any event, the pre-processing must be done in a way that doesn't use floating point math!!!** Often a combination of text manipulation along with integer math can do the job.