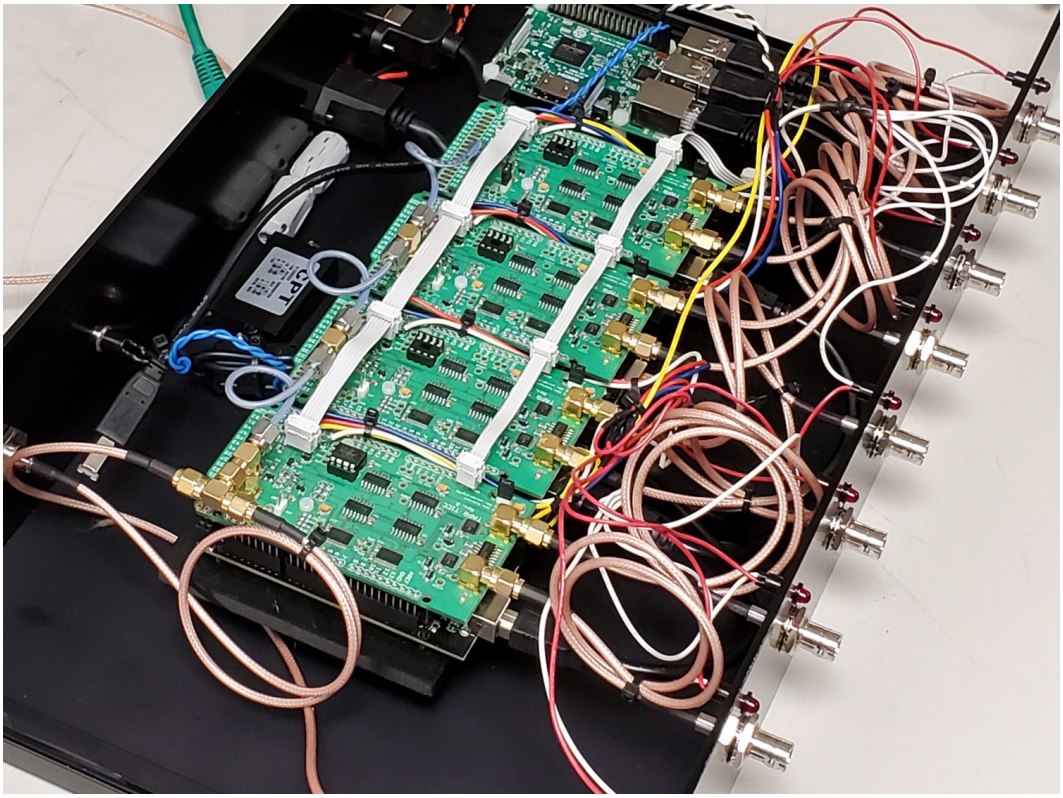
multi-TICC:
A Multi-Channel Timestamping Counter Based on the TAPR TICC

by John Ackermann N8UR[1]



The multi-TICC is an extension of the TAPR TICC[2] two-channel timestamping counter. It consists of multiple TICCs connected to each other and to a single board computer such as a Raspberry Pi. A prototype using four TICCs has been built and extensively tested. It allows timestamping up to eight input channels simultaneously, with the results made available via a network connection.

The multi-TICC can be thought of as a “timestamp appliance”: users can telnet to the unit and download live data from any or all of the channels. Input signals can be connected and disconnected at any time without impacting other channels. When a channel becomes active its timestamps may be directly compared with other channels.

---

1 jra@febo.com, https://febo.com
2 https://tapr.org/product/tapr-ticc/

If you are interested in assembling a multi-TICC, this application note and the files available at https://github.com/TAPR/TICC hopefully provide the information and design files you need.

The multi-TICC currently is **not** available as a TAPR product. The material cost is significant, as is the labor required for assembly and testing. We suspect that this is very much a niche device, and TAPR can't justify the financial and human resources required to stock it as a completed item. However, we know of sources potentially willing to build units to order. Please contact me if you're seriously interested in that.

A Bit About Timestamping

A **timestamping counter** uses a variable in its software program that increments in synchronization with an external **reference clock**. The variable represents the time elapsed since a fixed starting point (usually when the unit was powered on), and can be thought of as a **timescale** on which external events can be placed in time. Each event, typically the rising edge of a pulse-per-second ("**PPS**") signal output by the device under test ("**DUT**") is marked with its time of arrival (a "**timestamp**") on that timescale, and the sequence of timestamps is output via a communications port for further processing.

A timestamp is a low-level measurement and in most cases a single one is not of much value. But from a series of timestamps it is possible to derive phase, frequency, stability, and other information about an input signal compared to the reference clock. Time and frequency analysis software such as TimeLab[3] and Stable32[4] can read a sequence of timestamps and process that data to provide a wide range of information about the DUT.

A timestamping counter, like the TAPR TICC, that has more than one input channel can also be used for more complex measurements. For example, if a PPS signal from the DUT is applied to channel A, and a signal from another PPS source is applied to channel B, one can calculate the time interval between them simply by subtracting the timestamps. From that data one can also derive other useful measurements such as period and ratio. Timestamps are very useful building blocks.[5]

The two channels of the TICC can thus be used together for traditional time interval measurements, or independently to provide timestamps from two independent DUTs.

The multi-TICC Architecture

The TAPR TICC is a two-channel timestamping counter that consists of a "shield" mounted to an Arduino Mega 2560 controller. Firmware on the Arduino controls the system and provides output via its USB port. The TICC provides on-board headers that expose clock and synchronization signals, and by connecting these signals across two or more units, multiple

3 http://www.miles.io/timelab/readme.htm
4 https://ieee-uffc.org/frequency-control/frequency-control-software/stable32/
5 With the current TICC firmware, the timescale will not wrap for about 5.8 million years.

TICCs can share a single timescale. One TICC is set as the “host” in its configuration, and the others are set as “clients”. The host provides clock signals to the clients. In this way counters with four, six, eight, or even more inputs can be assembled, and one can compare measurements across all channels.

The USB output from each TICC is fed to one of the inputs of a single board computer such as a Raspberry Pi. Software on the computer reads the input streams and makes them available via an ethernet connection. The output can include data from one, many, or all of the TICC channels. Channels with no input signals are silently ignored until a signal appears.

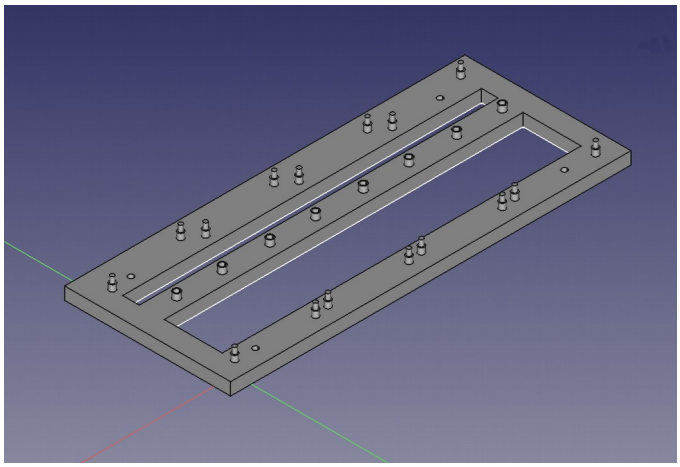
This means that the multi-TICC can be kept running across many measurement cycles, and measurements from multiple channels and multiple measurement cycles can be matched in sequence by their timestamps.[6] With the current TICC firmware, the timescale will not roll over for far longer than any of us need to worry about. Typically, only an interruption 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 test results.

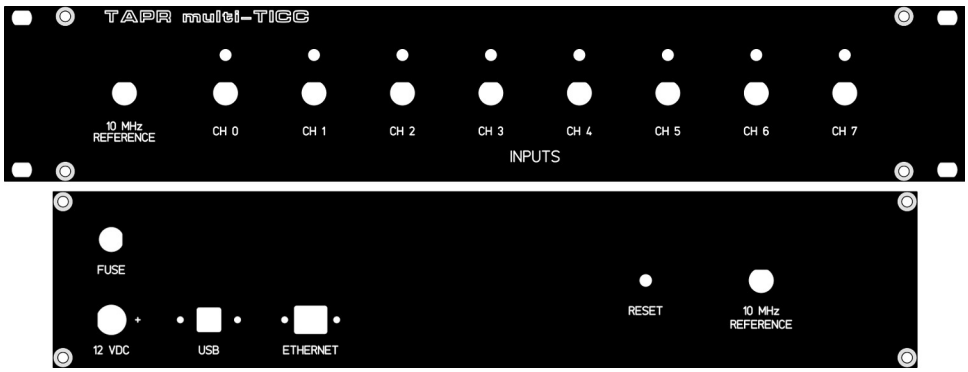
6 Enormous timestamps bring with them computational challenges. It’s often desirable to have timestamps “wrap”, or roll over like an odometer, to keep the value in range. In effect the value is truncated to two or three places. For example, when the timestamp reaches 100, the next sample might start again at 0. Most time stability analysis software tools can work data wrapped in this way.

Mechanical Design

Four TICCs flopping around on the workbench are a bit hard to manage. To make it much easier to work with the boards, I created[7] a 3-D printed carrier that will align and secure 4 boards. That carrier can in turn be mounted to a base plate.



If you want to make your multi-TICC all pretty, I've designed a complete 2U rack enclosure using the Front Panel Express[8] design tools.



The design files for the carrier and metal enclosure are available in the multi-ticc/enclosures directory of the GitHub repository.

7 With much help from Mike Suhar, W8RKO.
8 https://www.frontpanelexpress.com

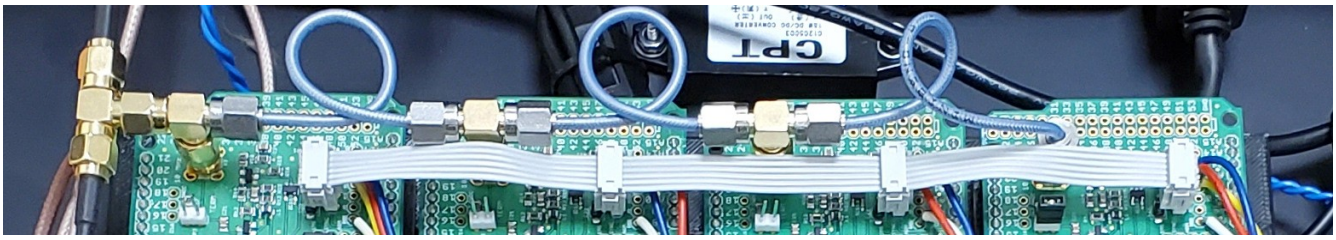
Wiring

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

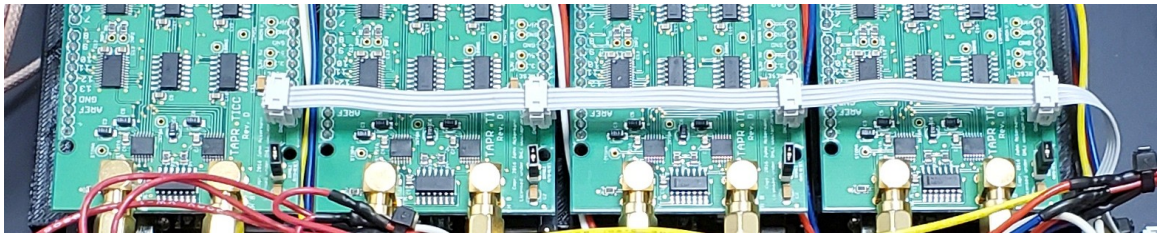
1. A common 10 MHz clock is applied to all boards through either a passive splitter or a distribution amplifier like the TADD-1. The “TERM” termination jumper should be installed on all boards. Note that if you use a passive splitter, you may need to increase the 10 MHz drive level to compensate for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

IMPORTANT NOTE: I originally used daisy-chained connectors (short SMA-SMA cables and SMA tee adapters) to pass one 10 MHz signal cable to the four TICCs, but that seemed to result in odd behavior, with boards showing differing noise levels; in the worst case, noise levels on one board would noticeably rise, or one board would show occasional phase jumps of several hundred picoseconds. There’s more discussion of this issue in the “Performance” section below, but testing has shown that feeding each board separately from a splitter or distribution amplifier, and setting the termination jumper on each board, avoids the problem.

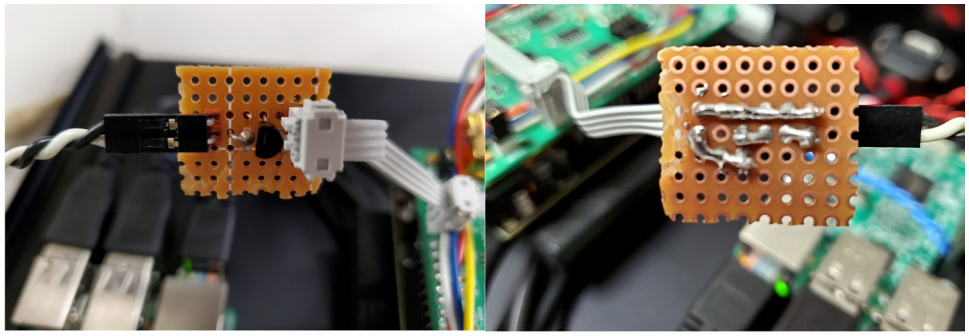
2. As shown below (but ignore the daisy-chained coax lines!), the three pins of the JP2 header are connected in a daisy-chain fashion across all the boards. The easiest way to make this cable is to use 0.1 inch spaced IDC ribbon cable connectors with 6 positions – just make sure you plug the same row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

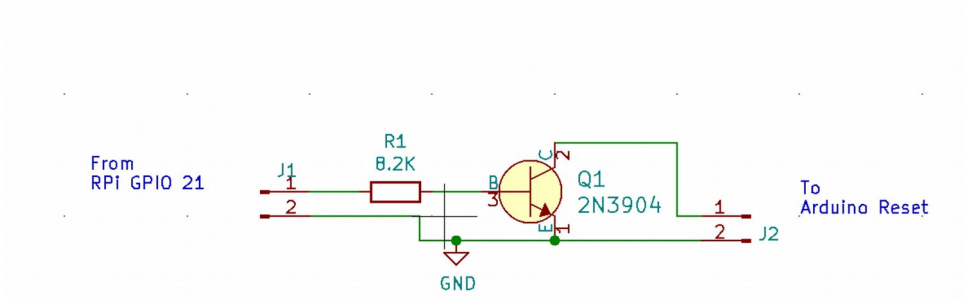


3. As shown below, the RESET lines of all the boards are also connected in parallel, so that they can be reset simultaneously. To do that, solder a 2 pin header in the holes marked “GNDB” and “RESET” located on the right front of each board, and make a daisy-chain connector tying them together. Again, it’s easiest to do this with a 4 pin ribbon cable. Allow extra length at one end if you want to allow the Raspberry Pi to provide a reset function as discussed in step 4 below. One note: with the reset lines in parallel, a powered-down TICC will load the reset line of the others, with the result that they will not start. Make sure all the TICCs are powered up when the reset lines are daisy-chained.



4. If a Raspberry Pi is used as the host computer, one of the Pi's GPIO pins can be used to cause the TICCs to reset via software command. The problem is that the Arduino processor board 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 removing the jumper or performing magic. The multi-ticc_updater.py program discussed below does the appropriate magic 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 "client" 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 client boards.

7. If you are building a front panel for the unit, each TICC has three connection points at A11, A12, and A13 for external LEDs. The LEDs should have a 270 to 1k resistor in series with the anode lead, and the cathode grounded. A11 is active when the system detects that

the required clock signals are present. A12 and A13 track the on-board LED1 and LED2, which indicate presence of signals on chA and chB respectively.

In the daisy-chain multi-TICC configuration, it makes sense to use the A11 signal on the last board of the chain as an overall “status OK” indicator since if that board is clocking properly, it is highly likely that the others are as well.

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

Firmware Configuration

The firmware shipped on recent TICC units[9] has basic multi-TICC capability included. However, a newer version (20200412.1, available in the TICC GitHub repository) adds helpful features for multi-TICC use:

1. You can assign channel IDs other than A and B via the [I] (“channel name”) configuration item.

2. You can use configuration item [J] (“PROP_DELAY”) to set an offset value, or propagation delay, 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.)

3. 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 configuration item [G] to set the master board (typically the first in the chain) as [H]ost, and to set the other boards as [C]lient.

Optionally, use the [I] item to change each 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 items to set the offset for each channel in picoseconds.

Note: 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.

9 version 20170309.1

Data Interface and Host Processor

If you have 4 TICCs 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, as well as some other tools. All are available under the multi-ticc/rpi_files directory of the TICC GitHub repository.

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 in the rpi_files directory is a description of software prerequisites that need to be installed to use the software, as well as some other setup information that 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, this program will look for four TICC units connected to the Raspberry Pi USB 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 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 sequentially update all connected TICCs with a new firmware file in hex format. Supply the file path and name as a command-line argument. (The program assumes all the TICCs have port names set to `/dev/ttyTICCx` as described above.)

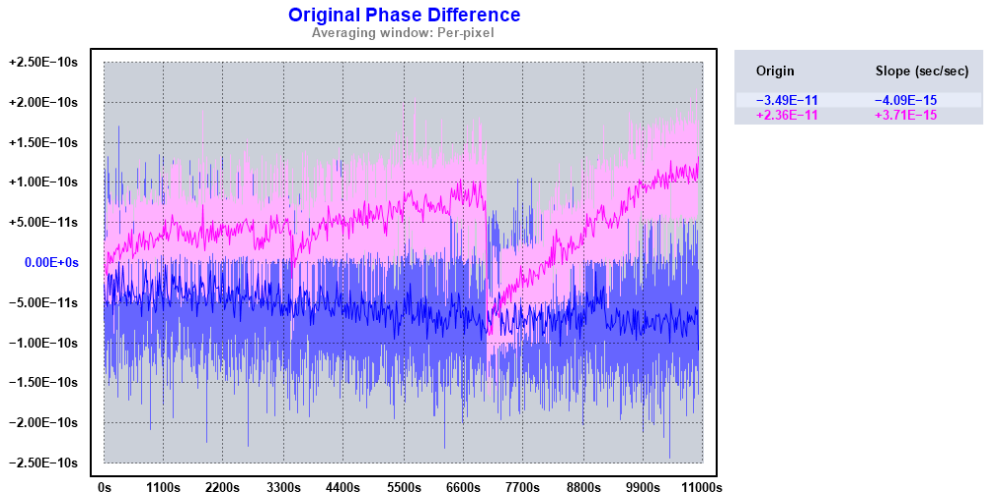
`miniterm.sh` is a shell script that will launch a simple terminal program allowing you to communicate with one TICC. Just give the TICC serial port name as a command line argument, such as `./miniterm.sh /dev/ttyTICC0`.

`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`.

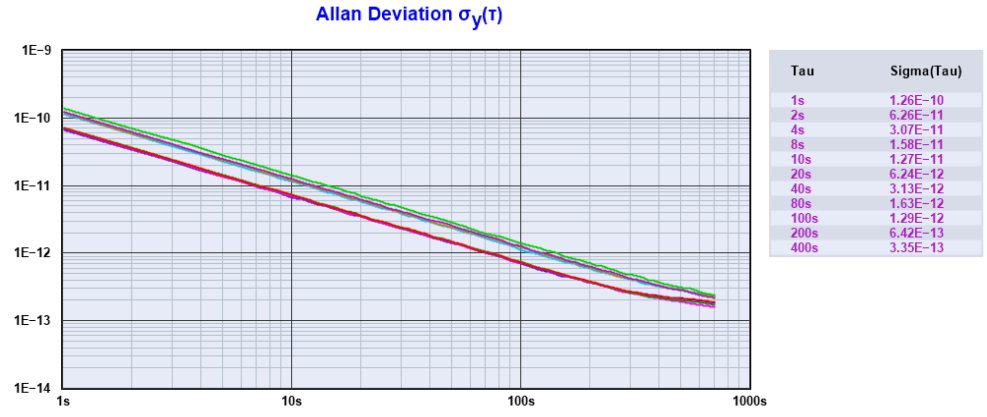
multi-TICC Performance

Performance tests on the multi-TICC indicate that very little if any performance is lost compared to a stand-alone TICC, and I've collected over 5 million samples on the prototype unit without any apparent glitches. 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 (short coax jumper cables with "tee" connectors on each TICC reference input) for the 10 MHz clock feed to the boards results in strange behavior. Either one or more of the boards shows higher or lower jitter than the others, or alternatively phase glitches can be introduced on one or more channels. Here are two examples from early testing.



Trace	Notes	Input Freq	Sample Interval	Acquired
chA (Unsaved)	Noise Floor via HP 5071A	1 Hz	1 s	10910 pts
chG (Unsaved)	Noise Floor via HP 5071A	1 Hz	1 s	10910 pts



Trace	Notes	Input Freq	Sample Interval	Acquired
chA (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chB (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chC (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chD (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chE (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chF (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chG (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts
chH (Unsaved)	Residual Noise	1 Hz	1 s	3160 pts

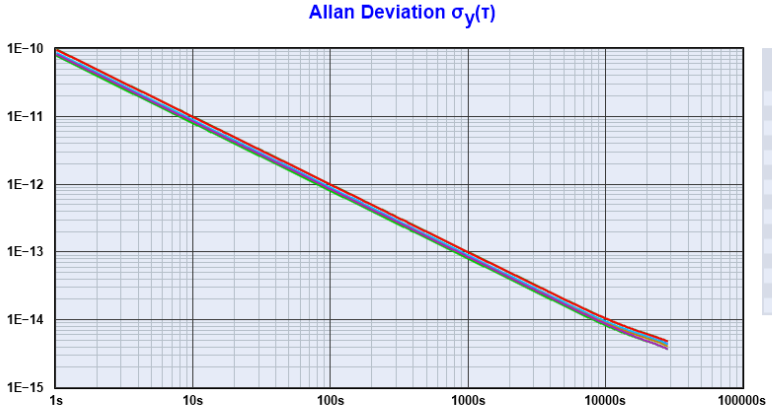
It is highly likely that the combination of multiple coax cables, “tee” connectors, and mismatch of the unterminated TICC clock input circuit caused reflections that translated into jitter at one or more of the 10 MHz reference inputs. Driving each TICC via its own cable from a passive splitter or distribution amplifier (remembering to install the termination jumper on each TICC, which ensures a good impedance match) appears to avoid the problem.

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 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 output from the 5071A went to a TADD-1 distribution amplifier, four of whose output channels were fed to one TICC 10 MHz reference input. The TADD-1 signal output was about 10 dBm..

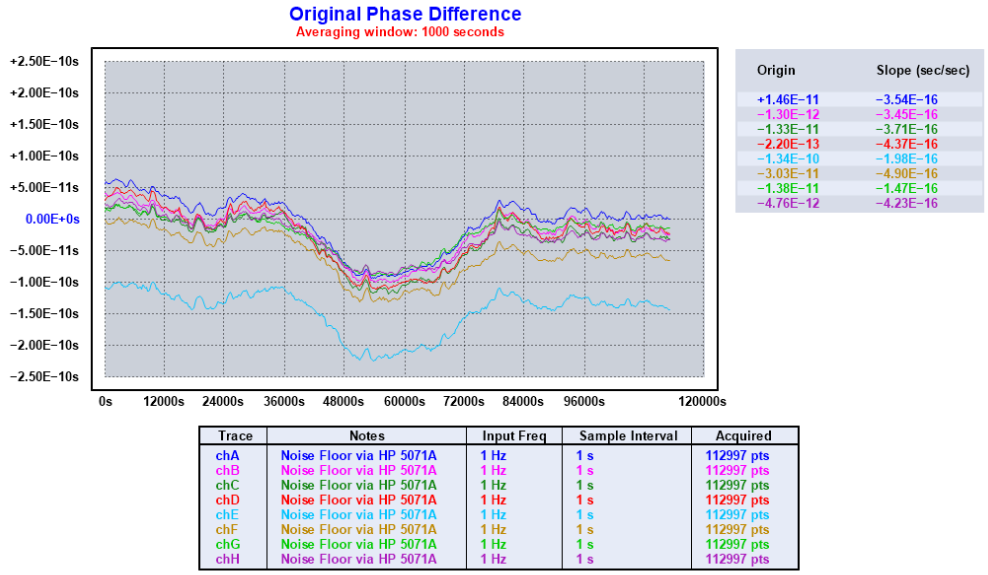
The 5071A PPS signal was fed into a 3dB attenuator and then to the multi-TICC inputs via a daisy-chain of cables using BNC tee connectors. At the end of the chain was a 50 ohm termination. The jumpers between channels each added about 2.4 nanoseconds delay. (While this daisy-chain could also result in reflection-induced jitter, none was noted in testing.)

With all that said, here are results from a test that collected over 125,000 samples from each of the 8 multi-TICC channels, 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

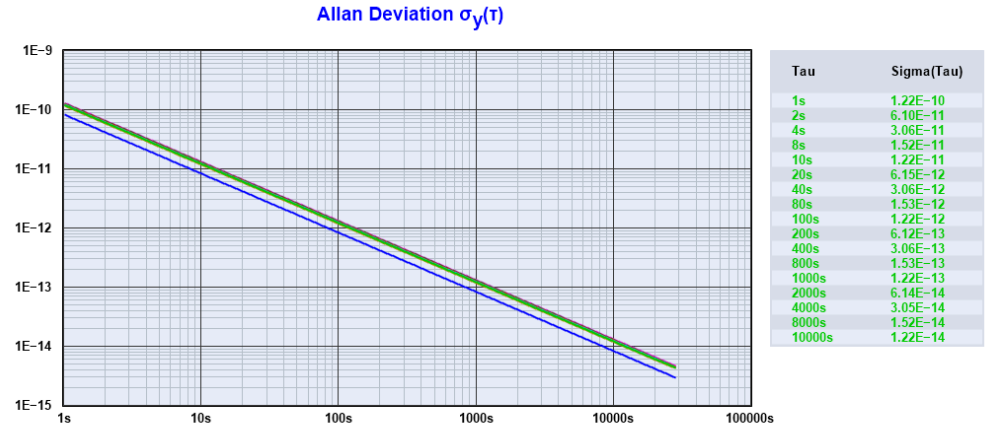
Trace	Notes	Input Freq	Sample Interval	Acquired
chA	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chB	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chC	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chD	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chE	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chF	Noise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



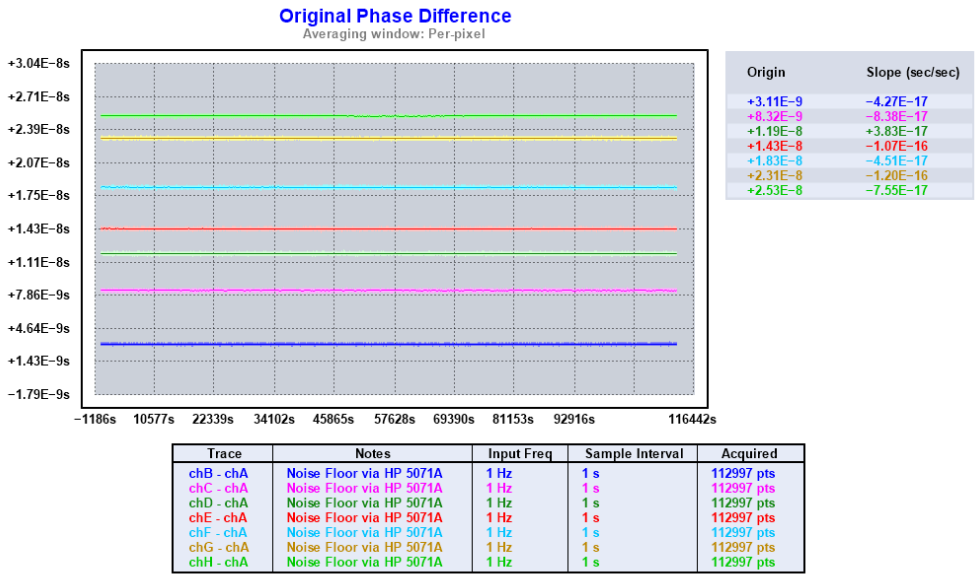
Trace	Notes	Input Freq	Sample Interval	Acquired
chA	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chB	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chC	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chD	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chE	Noise Floor via HP 5071A	1 Hz	1 s	112997 pts
chF	Noise 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 overnight temperature changes. Though it looks significant here, remember that the change is only about 100 picoseconds over about 40,000 seconds!

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.



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:

	ADEV	Picoseconds				
	@ 1 SEC	MEAN	SDEV	MIN	MAX	RANGE
chA	8.38e-11	149045	65.22	148827	149289	462
chB	8.64e-11	152151	64.50	151892	152438	546
chC	1.00e-10	157357	71.75	157056	157658	602
chD	9.72e-11	160956	72.76	160670	161244	574
chE	9.03e-11	163360	65.22	163121	163624	503
chF	8.56e-11	167386	63.05	167072	167632	560
chG	7.89e-11	172147	54.65	171917	172381	464
chH	8.35e-11	174307	57.62	174042	174525	483

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

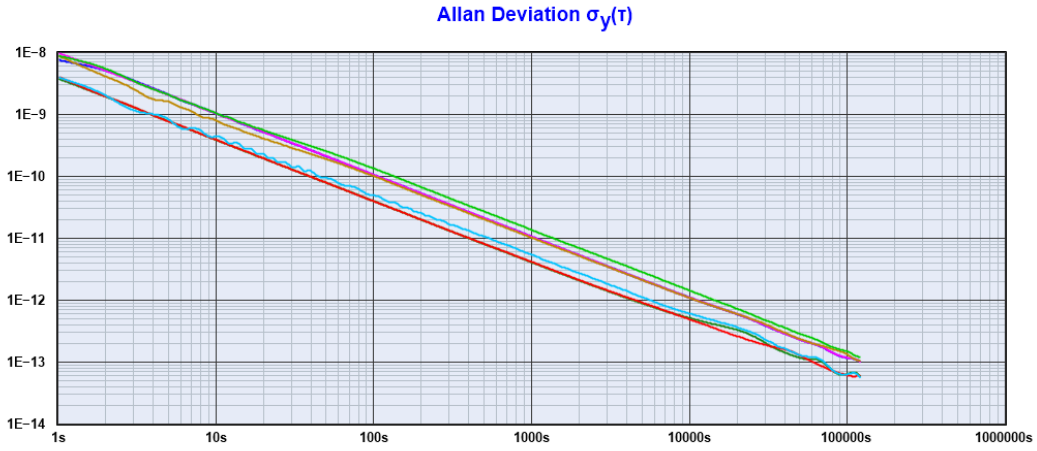
N = 112,970

Calculations from TVB ADEV1 and STAT

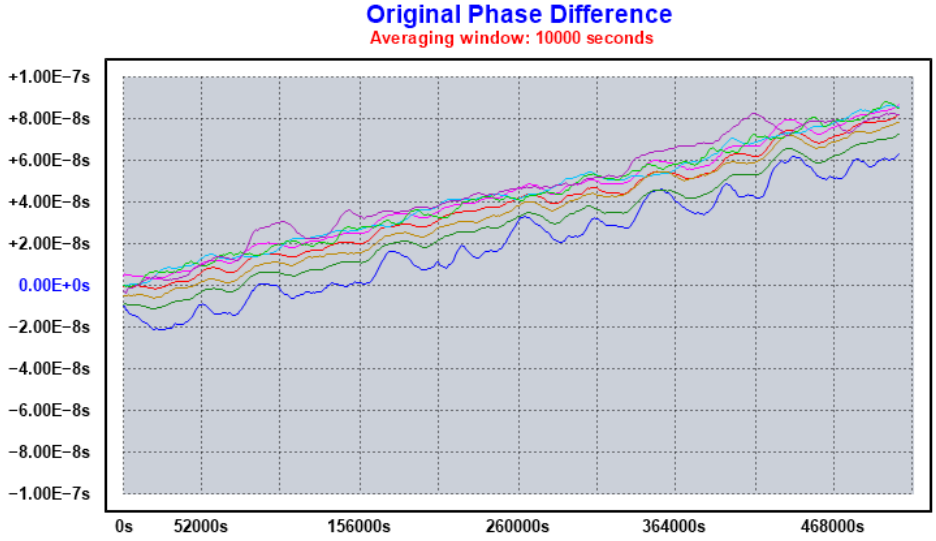
In the table above, the "chA" columns show the subtraction of channel A from each succeeding channel. The "Prev CH" columns show the subtraction of the prior channel (e.g., the "chC" row shows (chC – chB). The "Raw Values" are simple subtraction of the mean values reported above. The "Minus Cable Delay" 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, the following page shows an example of "real world" data capture from a multi-TICC with the PPS from eight GPS units connected to channels A through H:



Trace	Notes	Sample Interval	Acquired	Instrument
NEO-M8P	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M8T	vs HP 5071A	1 s	510244 pts	multi-TICC
ZED-F9P	vs HP 5071A	1 s	510244 pts	multi-TICC
ZED-F9T	vs HP 5071A	1 s	510244 pts	multi-TICC
LEA-M8F	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M9N	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M8N	vs HP 5071A	1 s	510244 pts	multi-TICC



Origin	Slope (sec/sec)
−2.09E−8	+1.66E−13
+2.38E−9	+1.61E−13
−1.14E−8	+1.61E−13
−2.15E−9	+1.61E−13
+2.83E−9	+1.61E−13
−6.82E−9	+1.66E−13
+2.09E−9	+1.62E−13
+5.23E−9	+1.62E−13

Trace	Notes	Sample Interval	Acquired	Instrument
CNS-II	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M8P	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M8T	vs HP 5071A	1 s	510244 pts	multi-TICC
ZED-F9P	vs HP 5071A	1 s	510244 pts	multi-TICC
ZED-F9T	vs HP 5071A	1 s	510244 pts	multi-TICC
LEA-M8F	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M8N	vs HP 5071A	1 s	510244 pts	multi-TICC
NEO-M9N	vs HP 5071A	1 s	510244 pts	multi-TICC