APS2 Documentation

Release 1.0

Blake Johnson, Colm Ryan, and Brian Donovan

Oct 08, 2018

Contents

This document serves as the user manual and programming guide for the Arbitrary Pulse Sequencer, version 2 (APS2).

Contents:

CHAPTER 1

Hardware Specifications

The BBN Arbitrary Pulse Sequencer 2 (APS2) is a modular system providing up to 18 channels of analog waveform generation with a maximum output rate of 1.2 GS/s and 14-bits of vertical resolution. Each module in an APS2 system provides two analog outputs, DC coupled into a fixed +/- 1V range, and four digital outputs (1.5 V) for triggering other equipment. Each APS2 module has 1 GB of DDR3 SDRAM for waveform and sequence storage, which is enough for over 64 million sequence instructions. A low-latency cache allows for fast access to 128K waveform samples. Each module can be independently triggered for sophisticated waveform scenarios.

The digital and analog circuits have been carefully engineered to provide extremely low-noise analog performance, resulting in a noise spectral density that is orders of magnitude lower than competing products, as shown in *Noise Comparison*.

1.1 Detailed Specifications

Analog channels	two 14-bit 1.2 GS/s outputs per module
Digital channels	four 1.5V outputs per module
Analog Jitter	7.5ps RMS
Digital Jitter	5ps RMS
Rise/fall time	2ns
Settling time	2ns to 10%, 10ns to 1%
Trigger modes	Internal, external, system, or software triggering
Ext. trigger input	1 V minimum into 50 Ω, 5 V maximum; triggered on *rising* edge
Reference input	10 MHz sine or square, 1V to 3.3V peak to peak (+4 to +14 dBm)
Waveform cache	128K samples
Sequence memory	64M instructions
Min instruction duration	8 samples
Max instruction duration	8M samples (~7ms at 1.2GS/s)
Max loop repeats	65,536

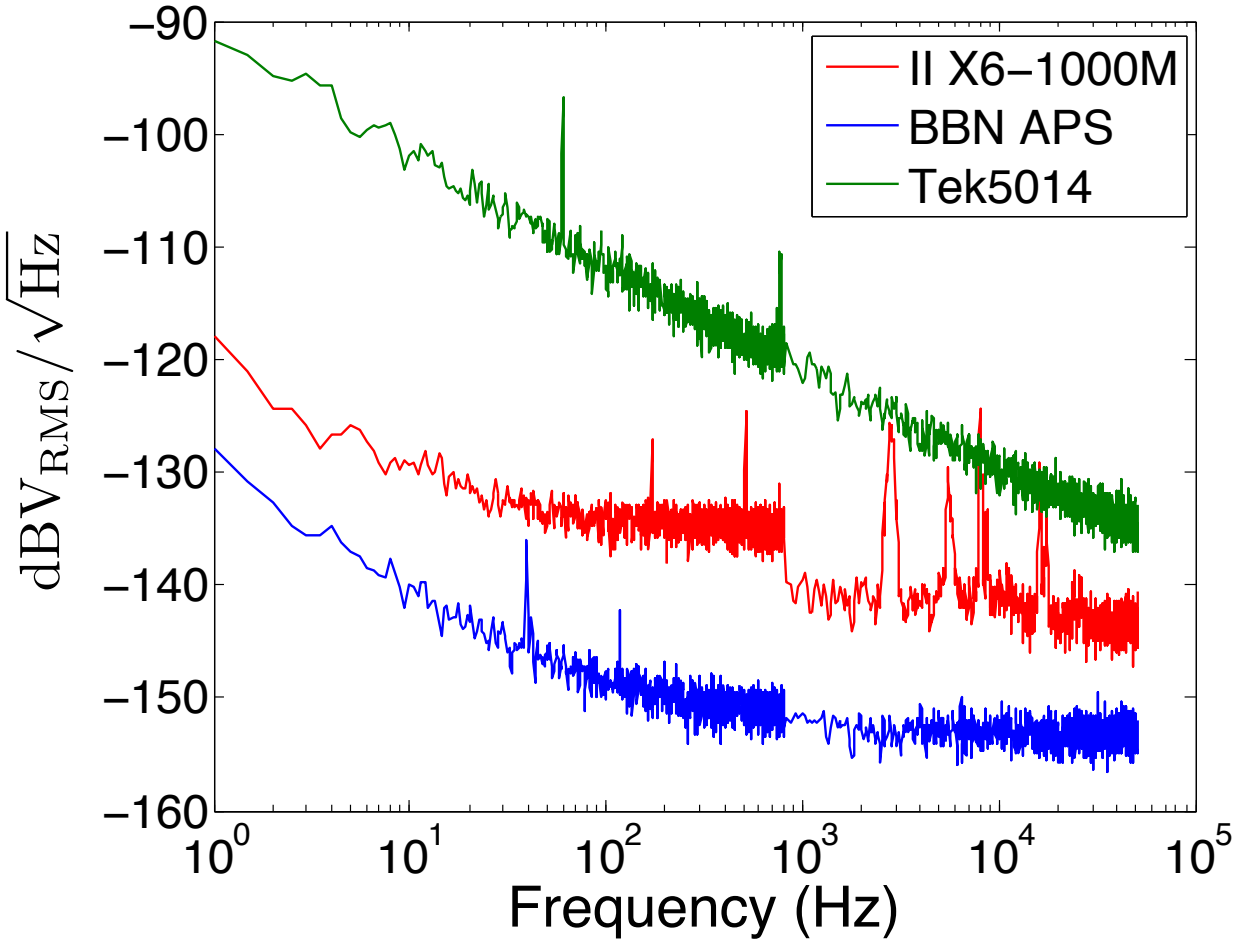


Fig. 1: **Comparison of AWG output noise** Output noise power versus frequency for the Tektronix AWG5014, Innovative Integration X6-1000M, and BBN APS. The APS's linear power supplies and low-noise output amplifier lead to signficant improvements in the noise performance. The II X6 is significantly better than the Tek5014, but suffers from resonances in the noise spectrum because it is in a host PC environment.

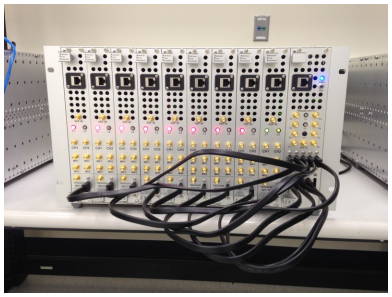


Fig. 2: **BBN APS2 front panel** The front panel of the APS has two analog outputs, 4 marker outputs, a trigger input, two SATA ports, a 1 GigE port, a 10 MHz reference input and two status LEDs.

1.2 Triggering

The APS2 supports four different types of triggers. The *internal* mode generates triggers on a programmable interval between 6.66ns and 14s. The *external* mode listens for triggers on the front-panel SMA "trigger input" port. In this mode, the APS2 is triggered on the rising edge of a 1-5V signal. The *system* trigger accepts triggers on the SATA input port from the APS2 Trigger Distribution Module (TDM). Finally, the *software* mode allows the user to trigger the APS2 via the host computer with the *trigger()* API method.

1.3 Communications Interface

The APS2 communicates with a host PC via UDP/TCP over 1GigE. 1GigE is required so ensure all switches between the host computer and APS2 support 1GigE. The APS2 supports static and DHCP assigned IP addresses. Instructions for setting the APS2 IP addresses are contained in the *Software* section.

1.4 Status LED's

The L1 and L2 LEDs provide status indicators for the communication (L1) and sequencing (L2) firmware components.

L1:

- off - SFP port failure
- green breathing - no ethernet connection;
- solid green - link established (but not necessarily connected to host);
- green blinks - receiving or transmitting data;
- red - fatal communication error. Power cycle the module to restore connectivity.

L2:

- dark - idle;
- solid green - playback enabled and outputing sequences;
- green breathing - playback enabled but no trigger received in the past 100ms;
- solid red - fatal cache controller error. Power cycle the module to restore playback functionality.
- blinking red - cache stall in playback. See cache for details.

CHAPTER 2

Installation Guide

2.1 Hardware

IMPORTANT NOTE: The APS2 enclosure has mounting brackets on the sides for securing the box to a standard 19" rack. However, these brackets *cannot* support the weight of the entire APS2 system. Therefore, you must mount the APS2 on a shelf. Furthermore, the top-rear section of the APS2 enclosure has a screen which serves as the air intake for the cooling system. Never stack equipment on top of the APS2 in a way which blocks this air intake.

The BBN APS2 system contains one or more analog output modules and an trigger distribution module in an enclosure that supplies power to each module. Up to 9 analog modules may be installed in a single 19" 8U enclosure, providing 18 analog output channels. Installing a new module only requires plugging it into a free slot of a powered-off system, then optionally connecting a SATA cable from the new APS module to the trigger module. The SATA interface is required to send signals between the APS and TDM modules.

Each module in an APS2 system acts as an independent network endpoint. The modules communicate with a host computer via a UDP/TCP interface over 1GigE. The APS2 will not auto-negotiate down to 100Mb or 10Mb so you must connect via a gigabit switch and appropriate patch cable (cat 5e or cat 6). To ensure high-bandwidth throughput, it is important that the APS2 and the host computer are not separated by too many network hops. If possible, locate the host and APS2 on a common switch or router[1].

While the APS2 can run in a standalone clock configuration, to achieve synchronous output from multiple modules requires supplying a 10 MHz (+7 dBm) external reference (square wave or sine wave) to *each* module at the corresponding front panel input. Multiple devices are then synchronized with a phase-synchronous external trigger delivered to each module, or by using the TDM to generate a system trigger.

2.2 Software

The APS2 control interface is provided by a C-API shared library (libaps2) as as well as wrappers in common scientific programming languages (python, MATLAB, and Julia). Pre-built binaries for the shared library are available for several platforms, including Win64, linux, and macOS. A demonstration script and several example sequence files are

[1] The APS2 typically uses static self-assigned IP addresses and should ideally be behind the same router as the control computer.

available in the libaps2 source code repository. Therefore, we recommend that users download the latest version of the source from GitHub (http://github.com/BBN-Q/libaps2).

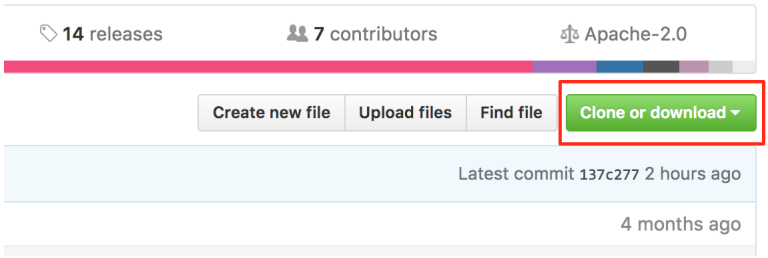


Fig. 1: Download a ZIP file with the latest libaps2 source code by clicking on the link shown above.

The GitHub website provides a link to download a zip file with the latest snapshot of the source repository. Git users may instead clone the repo with:

```
git clone https://github.com/BBN-Q/libaps2.git
```

Using git has the advantage that users may easily upgrade to future libaps2 releases with `git pull`. Description of the source repository files is available in the *File list* section.

2.2.1 Installation via conda

Users of the Anaconda python distribution can install libaps2 via `conda`:

```
conda install -c bbn-q libaps2
```

This command will install the libaps2 shared library and libaps2 python wrapper into your Anaconda path (`~/anaconda3/lib` on posix-systems, or `C:\Users\%USERNAME%\Anaconda3\Library` on windows) such that it can be easily loaded by other programs. To verify that it worked, launch an ipython REPL and try:

```
In [1]: import aps2
```

If that command runs without errors, the install completed successfully. If you encounter errors, verify that the Anaconda library folder is on your PATH.

2.2.2 Installation from source

Users that need ultimate control over the libaps2 dependencies will want to build the shared library from source. Follow the instructions in the README to build the binaries. Our cmake scripts include an `install` target which will install the binaries in appropriate locations on your system with:

```
make install
```

The libaps2 python wrapper can be installed separately with:

```
cd src/python
pip install .
```

To verify successful installation, try importing libaps2 into a python console, as shown above.

2.2.3 Manual installation

A third option is to download pre-built binaries from the "Releases" tab on GitHub (http://github.com/BBN-Q/libaps2/releases). Download the archive corresponding to your operating system and extract the files. On posix-systems, copy the contents of the archive to corresponding locations in `/usr/local` (i.e. copy files from `lib/` to `/usr/local/lib` and `include/` to `/usr/local/include`). On windows, add the `bin` folder to the PATH variable, or copy `libaps2.dll` to `C:\windows\system32`.

2.2.4 File list

The releases follow a directory structure that corresponds to the git repository.

- ***examples* - Example sequence and waveform files**
  - *aps2_demo.m* - Matlab demonstration script
  - ***aps2_demo.py* - Python demonstration script**
    - a full scale ramp;
    - gaussian pulses from 256 samples down to 8 samples with 10ns gaps;
    - square wave from 256 down to 8 samples with 10ns gaps;
    - wfB.dat is negative wfA.dat.
  - *cpmg.h5* - a CPMG sequence *Y90 - (delay - X180 - delay)^n - Y90m* with n = [4, 8, 16, 32, 64]
  - *instr_prefetch.h5* - demonstration of subroutine prefetching
  - *ramsey.h5* - a Ramsey sequence *X90 - delay - X90m*
  - *ramsey_tppi.h5* - a Ramsey experiment with the second pulse phase modulated by Time Proportional Phase Increment using the *PHASE_OFFSET* instruction
  - *ramsey_tppi_ssb.h5*- same as *ramsey_tppi* but with SSB modulation of the pulses using on-board modulation.
  - *ramsey_slipped.h5* - a Ramsey pattern but with the markers slipped by one sample to show the marker resolution and jitter.
  - *wfA.dat/wfB.dat* - test waveform patterns for *play_waveform* executable as signed integers one sample per line:
- ***src* - the source code**
  - *src/lib* - the shared library. `libaps2.h` contains the public API definitions.
  - *src/matlab* - Matlab bindings to libaps2
  - *src/julia* - Julia bindings to libaps2
  - *src/python* - python bindings to libaps2
  - *src/util* - test and utility command line programs. See below for description.
  - *src/C++* - C++ command line programs to play waveforms and sequences.
  - *src/wireshark* - lua dissector for sniffing APS2 packets.
- ***build* - compiled shared library and executable programs**
  - **Shared library**
    - *libaps2.dll* - the main shared library

- **Command line programs**
  - *aps2_play_waveform.exe* - command line program to play a single waveform on the analog channels.
  - *aps2_play_sequence.exe* - command line program to play a HDF5 sequence file.
- **Command line utilities**
  - *aps2_enumerate.exe* - get a list of APS2 modules visible on the network subnet.
  - *aps2_program.exe* - update the firmware. See *Firmware Updates*.
  - *aps2_flash.exe* - update IP/DHCP and MAC addresses and the boot chip configuration sequence.
  - *aps2_reset.exe* - reset an APS2.
- **Self-test programs**
  - *aps2_run_tests.exe* - runs the unit test suite

2.2.5 Writing Sequences

The BBN APS2 has advanced sequencing capabilities. Fully taking advantage of these capabilities may require use of higher-level languages which can be 'compiled down' into sequence instructions. BBN has produced one such language, called Quantum Gate Language (QGL, http://github.com/BBN-Q/QGL), as well as a parameter management GUI in the PyQLab suite (http://github.com/BBN-Q/PyQLab). We encourage end-users to explore using QGL for creating pulse sequences. You may also find the sequence file export code to be a useful template when developing your own libraries. A detailed instruction format specification can be found in the *Concrete Instructions* section.

2.3 Networking Setup

Once the APS2 has been powered on, the user may assign static IP addresses to each module. By default, the APS2 modules will have addresses on the 192.168.2.X subnet (e.g. the leftmost module in the system will have the address 192.168.2.2, and increase sequentially left-to-right). The `enumerate()` method in libaps2 may be used to find APS2 modules on your current subnet. Another method, `set_ip_addr()` or the `aps2_flash` utility may be used to program new IP addresses. Since the APS2 modules will respond to any valid packet on its port, we recommend placing the APS2 system on a private network, or behind a firewall. The APS2 can also be setup to obtain a dynamically assigned IP address from a DHCP server. The `aps2_flash` utility can be used to toggle between static and dynamic but the APS2 must be reset or power cycled for the setting to take effect. If the DHCP look-up fails the system will fall back to its static IP address.

The control computer must be on the same subnet as the APS2 to respond to returning packets. Most operating systems allow multiple IP addresses to coexist on the same network card so the control computer can add a virtual IP on the APS2 subnet.

2.3.1 Windows

Under the Control Panel - Network and Internet - Network Connections click on the "Local Area Connection" and then properties to change the adapter settings. Then set the properties of the TCP/IPv4 interface.

Then under the Advanced tab it will be possible to add additional IP addresses. Unfortunately, Windows does not support multiple IP addresses with DHCP so a static address is required for the main network.

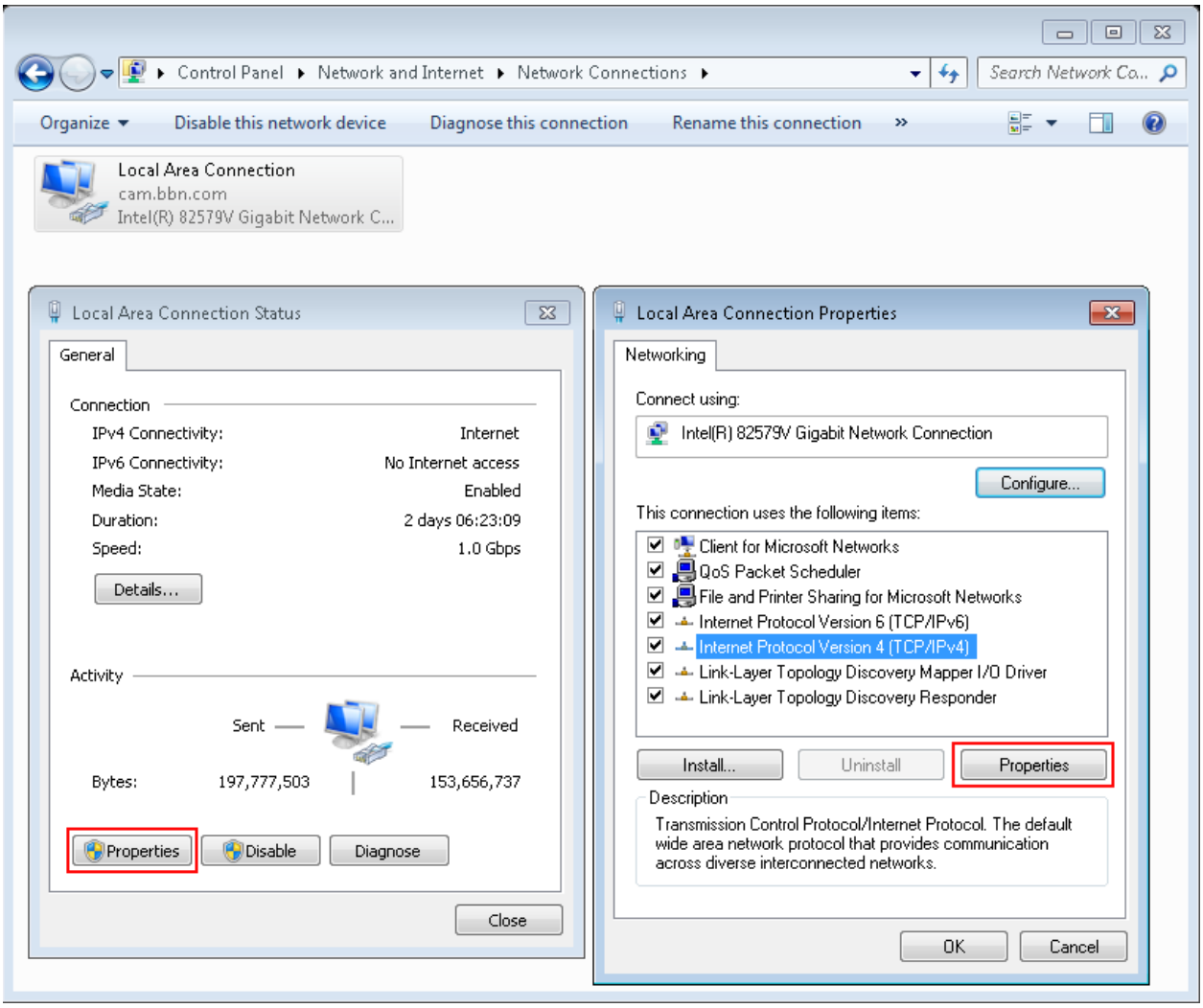


Fig. 2: **Step 1** accessing the IPv4 settings for the network interface.

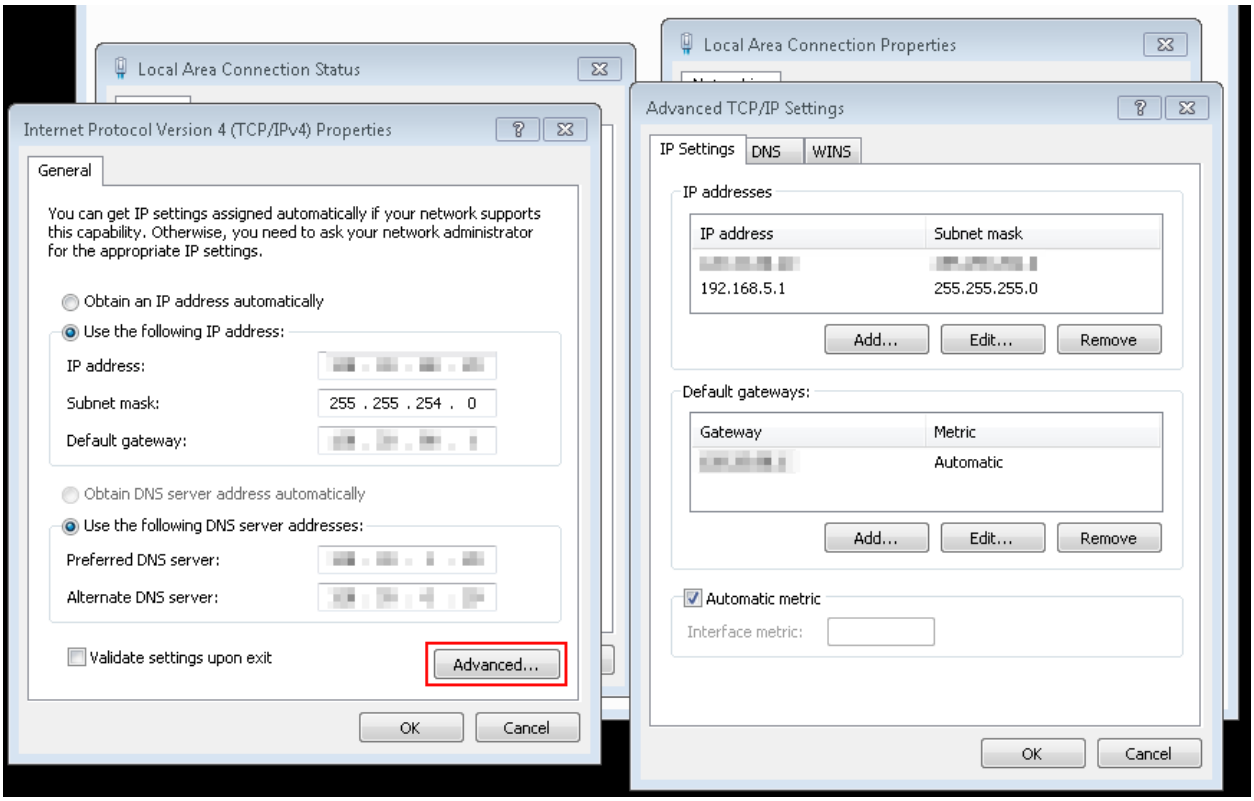


Fig. 3: **Step 2** Adding addition IP addresses for the network interface.

2.3.2 Linux

Temporary IP addresses can be obtained by adding additional ethernet interfaces using the *ip* command:

```
sudo ip addr add 192.168.2.29/24 dev eth0
```

A more permanent solution would involve editing the network interfaces file, e.g. `/etc/network/interfaces`.

2.3.3 OS X

In the System Preferences pane under Networking use the "Plus" button to add an interface.

2.4 Firmware Updates

BBN releases periodic firmware updates with bug-fixes and enhancements. These can be loaded onto the APS2 modules using the `aps2_program` executable:

```
./aps2_program
BBN AP2 Firmware Programming Executable
USAGE: aps2_program [options]

Options:
  --help      Print usage and exit.
```

(continues on next page)

(continued from previous page)

```
  --bitFile   Path to firmware bitfile.
  --ipAddr    IP address of unit to program (optional).
  --progMode  (optional) Where to program firmware DRAM/EPROM/BACKUP (optional).
  --logLevel  (optional) Logging level level to print (optional; default=2/INFO).

Examples:
  program --bitFile=/path/to/bitfile (all other options will be prompted for)
  program --bitFile=/path/to/bitfile --ipAddr=192.168.2.2 --progMode=DRAM
```

The executable will prompt the user for IP address and programming mode. The APS2 can boot from multiple locations: volatile DRAM; non-volatile flash or if all else fails a master backup in flash. The DRAM storage takes only a few seconds to program and is used for temporary booting for testing purposes. It will be lost on a power cycle. Once you are happy there are no issues with the new bitfile you can program it to the flash memory so the module will boot from the new firmware on a power cycle. This process involves erasing, writing and verifying and takes several minutes. The backup firmware should only be programmed in the rare case BBN releases an update to the backup image. Should something catastrophic happen during programming (unplugging the ethernet cable) the module may drop to the backup image which has a fixed IP of 192.168.2.123.

CHAPTER 3

Pulse Sequencing

3.1 Background

Sequencing typically requires construction of a sequence table which defines the order in which waveforms are played along with control-flow instructions. In existing commercial AWGs, these control-flow instructions are limited to repeated waveforms (basic looping) and non-conditional goto statements to jump to other sections of the waveform table. More recently, equipment manufacturers have added rudimentary conditional elements through *event triggers* to conditionally jump to an address in the waveform table upon receipt of an external trigger. This capability introduces *branches* into the sequence table. Equipment manufacturers have also expanded the memory re-use concept by *subsequences* which allow for jumping to sections of the waveform table and then returning to the jump point in a manner similar to a subroutine or function call in a standard programming language.

These recent additions expand the number of sequence flow graphs that can be built with these primitives. However, they are still limited in several ways. First, previous implementations have not allowed arbitrary combinations of control-flow constructs. For instance, it may be desirable to have *all* control-flow instructions be conditional, so that, for example, subsequence execution could depend on external input. Or it may be desirable to construct recursive control-flow structures, i.e. nested subsequences should be possible. Second, *event triggers* are not sufficiently expressive to choose between branches of more than two paths. With wider, multi-bit input interfaces, one can construct higher-order branches (e.g. with a 2-bit input you could have four choices).

In short, rather than having an instruction set that allows for a limited number of control-flow graphs, we wish to expand the instruction set of AWGs to allow for fully arbitrary control flow structures.

3.2 Superscalar Sequencer Design

To achieve arbitrary control flow in an AWG, we adopt modern CPU design practice and separate the functions of control flow from instruction execution. An instruction decoder and scheduler dispatches waveform and marker instructions to independent waveform, marker and modulation engines, while using control-flow instructions to decide which instruction to read next. This asynchronous design allows for efficient representation of common AWG sequences. However, it also requires reintroducing some sense of synchronization across the independent waveform and marker engines. This is achieved in two ways: SYNC instructions and write flags. The SYNC instruction ensures that all execution engines have finished any queued instructions before allowing sequencer execution to continue. The

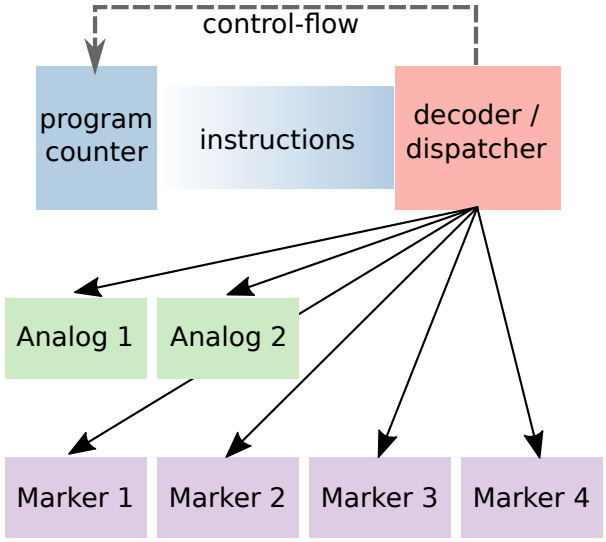


Fig. 1: **Sequencer block diagram** The APS has a single instruction decoder that dispatches instructions to multiple waveform and marker engines.

write flag allows a sequence of waveform and marker instructions to be written to their respective execution engines simultaneously. A waveform or marker instruction with its write flag low will be queued for its corresponding execution engine, but instruction delivery is delayed until the decoder receives an instruction with the write flag high.

3.3 Cache Design

The deep DDR3 memory comes with a latency penalty, particularly when jumping to random addresses. Rather than flowing these constraints down to the sequencer, the APS2 attempts to hide the latency by caching instruction and waveform data on board the sequencing FPGA. The cache follows some simple heuristics and needs some hints in the forms of explicit *PREFETCH* commands for more sophisticated sequencing.

3.3.1 Instruction Cache

The APS2 instruction cache is split into two parts to support two different heuristics about how the sequence will move through the instruction stream. Both caches operate with cache lines of 128 instructions. The first cache is a circular buffer centered around the current instruction address that supports the notion that the most likely direction is forward motion through the instruction stream with potentially jumps to recently played addresses when looping. The controller greedily prefetches additional cache lines ahead of the current address but leaves a buffer of previously played cache lines. Function calls do not fit into these heuristics so the second part of the cache is a fully associative to support jumps anywhere to subroutines. The subroutine caches are filled in round-robin fashion with explicit *PREFETCH* instructions. The controller will ignore *8PREFETCH* instructions where the line is already in the cache. If the sequencer asks for an address not in either cache the cache will flush the circular buffer and fetch the aligned line into the start of the circular buffer. The sequencer LED will blink 1 per second to indicate the cache stall.

3.3.2 Waveform Cache

The waveform cache can hold a maximum of 131072 (128k) samples. When the cache is enabled, the cache preloads the first 128k samples from waveform memory. If the waveform library is smaller than this then nothing further is needed. To support sequences that need deeper waveform memory, the cache is split into two to enable bank bouncing

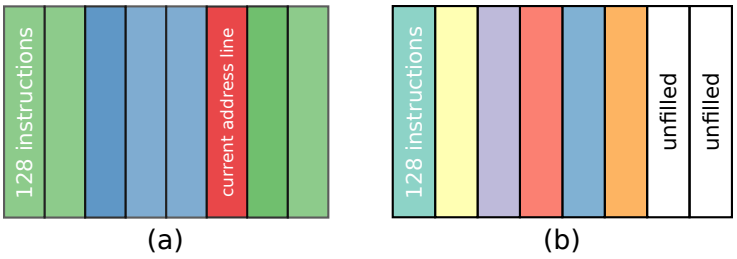


Fig. 2: **Instruction cache architecture** The instruction cache has two part: (a) a circular buffer centered around the currently playing address and (b) a fully associative cache for subroutine calls.

between a playing and a loading section. Loading is triggered by explicit waveform engine *PREFETCH* commands which will load 64k samples at an address aligned to a 64k sample boundary. Due to the vagaries of SDRAM accesses the time for this prefetch varies but should be approximately 200μs. Should the waveform engines ask for an address not in the waveform cache the cache controller will flush fetch the aligned 64k sample segment into the first half. The sequencer LED will blink twice per second to indicated the cache miss.

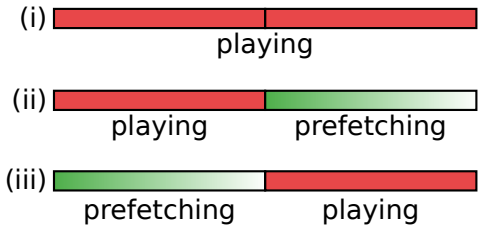


Fig. 3: **Waveform cache architecture** The waveform cache can be used to simply play waveforms from the first 128k samples (i) or with explicit waveform *PREFETCH* commands can be used in bank-bouncer mode (ii) and (iii).

3.4 Waveform Modulation

When an APS2 slice is used to drive the I and Q ports of an I/Q mixer to amplitude and phase modulate a microwave carrier it is convenient to bring some features typically baked into the waveforms back into the hardware. For example, if the I/Q mixer is used to single-side-band modulate the carrier the hardware can track the phase evolution through non deterministic delays and the phase modulation can be updated as part of the sequence. This can then even occur conditionally as part of the sequence control flow.

To support both the SSB modulation and dynamic frame updates, the APS2 can dispatch instructions to the modulation engine. The modulation engine controls the modulator which has up to four (current firmware has two) numerically controlled oscillators (NCO). When selected, the NCO with a phase Θ rotates the (a,b) output pair to (a cosΘ + b sinΘ, b cosΘ - a sinΘ). Multiple NCOs are supported to enable merging multiple logical channels at different frequencies onto the same physical channel. The modulation engine supports the following instructions

WAIT stall until a trigger is received

SYNC stall until a sync signal is received

RESET PHASE reset the phase of the selected NCO(s)

SET PHASE OFFSET set the phase offset of the selected NCO(s)

SET PHASE INCREMENT set the phase increment of the selected NCO(s) which sets an effective frequency

UPDATE FRAME update the frame of the selected NCO(s) by adding to the current frame

MODULATE apply modulation using the selected NCO for a given number of samples

All NCO phase commands are held until the the next boundary which is the end of the currently playing *MODULATE* command or a trigger/sync signal being received. The commands are held to allow them to occur at specific instances. For example, we want the phase to be reset at the trigger or the Z rotation implemented as a frame change to occur at the end of a pulse.

In addition, to account for mixer imperfections that can be inverted by appropriate adjustments of the waveforms the APS2 applies a 2x2 correction matrix applied to the I/Q pairs followed by a DC shift.

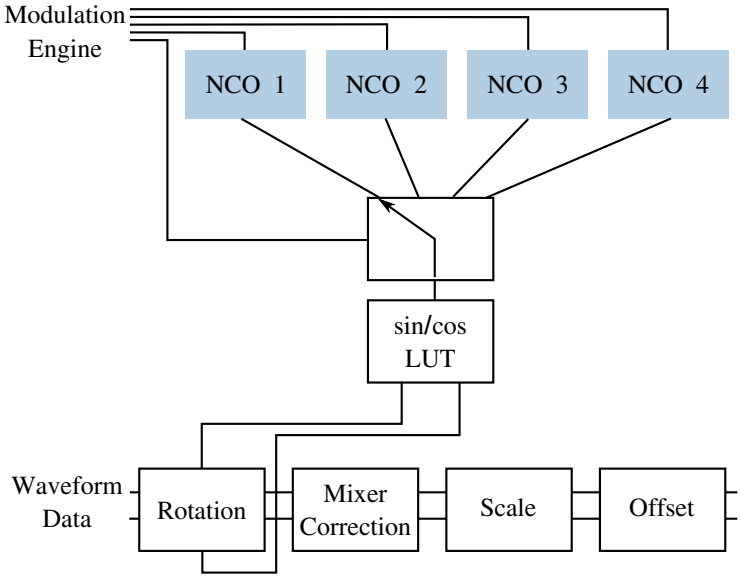


Fig. 4: **Modulator block diagram** Block diagram of the on-board modulation capabilities. The NCOs phase accumulators are controlled by the modulation engine which can also choose which NCO to select on a pulse by pulse basis. The selected phase is used for a sin/cos look up table (LUT) which provides values for the rotation matrix. The waveform pairs are subsequently processed through of arbitrary 2x2 matrix for amplitude and phase imbalance, channel scaling and offset.

CHAPTER 4

Trigger Distribution Module

The trigger distribution module (TDM) provides a flexible mechanism for distributing triggers and pulse sequence steering information across an APS2 crate. Since we expect our users to have diverse requirements for distributing steering information, we have decided to deliver the TDM as a reconfigurable device with a basic firmware that will satisfy certain needs.

4.1 Base Functionality

The base firmware delivered with the TDM will distribute signals captured on its front-panel interface to all APS2 modules connected by SATA cables. Port T8 is used as a 'valid' signal to indicate that data is ready to capture on T1-T7. On the rising of a signal on T8, the signals on T1-T7 are captured into a 7-bit trigger word which is immediately distributed across the APS2 crate. Each of the trigger ports T1-T8 drives a comparator with a programmable threshold. The base firmware fixes this threshold at 0.8V.

4.2 Building Custom Firmware

The TDM firmware source may be found here: https://github.com/BBN-Q/APS2-TDM

To get starting creating your own APS2 TDM firmware, you need a copy of Xilinx Vivado 2015.1 (the free Webpack edition is sufficient). Note that there are more recent versions of Vivado, but that the firmware source refers to specific versions of Xilinx IP cores, and it may be necessary to convert these for use with later Vivado versions. The APS2 TDM firmware relies upon one not-free Xilinx IP Core, the Tri-Mode Ethernet Media Access Controller, or TEMAC: http://www.xilinx.com/products/intellectual-property/temac.html

You can build the firmware without buying a TEMAC license, but the controller will stop functioning ~8 hours after loading the image, and you will be forced to power cycle the APS2 TDM to continue. We recommend purchasing a project license for the Xilinx TEMAC if you plan to build your own TDM firmware.

Refer to the README in the APS2-TDM firmware source for instructions on creating a Vivado project.

CHAPTER 5

APS2 Instruction Set

5.1 Abstract Instructions

Arbitrary control flow requires three concepts: sequences, loops (repetition) and conditional execution. We add to this set the concept of subroutines because of their value in structured programming and memory re-use.

The BBN APS2 has two memories: a *waveform* memory and an *instruction* memory. These memories are accessed via an intermediate caching mechanism to provide low-latency access to nearby sections of memory. In addition, the APS has four other resources available for managing control flow: a repeat counter, an instruction counter, a stack, and a comparison register. The instruction counter points to the current address in instruction memory. The APS2 sequence controller reads and executes operations in instruction memory at the instruction pointer. Unless the instruction specifies otherwise, by default the controller increments the instruction pointer upon executing each instruction. The available *abstract* instructions are:

SYNC
WAIT
WAVEFORM *address N*
MARKER *channel state N*
LOAD_REPEAT *count*
REPEAT *address*
CMP *operator N*
LOAD_CMP
GOTO *address*
CALL *address*
RETURN
MODULATOR
PREFETCH *address*
NOOP

We explain each of these instructions below.

SYNC—Halts instruction dispatch until waveform and marker engines have executed all queued instructions. The write flag should be set to broadcast this instruction.

WAIT—Indicates that waveform and marker engines should wait for a trigger before continuing. The write flag should be set to broadcast this instruction.

WAVEFORM—Indicates that the APS should play back length *N* data points starting at the given waveform memory address. An additional flag marks the waveform as a time-amplitude variant, which outputs data at a *fixed* address for *N* counts.

MARKER—Indicates the APS should hold marker *channel* in *state* (0 or 1) for *N* samples.

LOAD_REPEAT—Loads the given value into the repeat counter.

REPEAT—Decrements the repeat counter. If the resulting value is greater than zero, jumps to the given instruction address by updating the instruction counter.

CMP—Compares the value of the comparison register to the mask *N* with any of these operators: $=, \neq, >, <$. So, (CMP $\neq$ 0) would be true if the comparison register contains any value other than zero.

LOAD_CMP—Loads the comparison register with the next value from the message queue (received from the trigger distribution module). If the queue is empty, this instruction will halt execution until a message arrives.

GOTO—Jumps to the given address by updating the instruction counter.

CALL—Pushes the current instruction and repeat counters onto the stack, then jumps to the given address by updating the instruction counter.

MODULATOR—A modulator instruction.

RETURN—Moves (pops) the top values on the stack to the instruction and repeat counters, jumping back to the most recent CALL instruction.

PREFETCH—Prefetches a cache line of instructions starting at address

NOOP—Null or No Operation

These instructions easily facilitate two kinds of looping: iteration and while loops. The former is achieved through use of LOAD_REPEAT to set the value of the repeat counter, followed by the loop body, and terminated by REPEAT to jump back to the beginning of the loop. The latter is achieved by a conditional GOTO followed by the loop body, where the address of the GOTO is set to the end of the loop body.

Subroutines are implemented with the CALL and RETURN instructions. The address of a CALL instruction can indicate the first instruction in instruction memory of a subroutine. The subroutine may have multiple exit points, all of which are marked by a RETURN instruction.

Conditional execution is directly supported by the GOTO, CALL, and RETURN instructions. When these instructions are preceeded by a CMP instruction, their execution depends on the comparison resulted. Consequently, the stated instruction set is sufficient for arbitrary control flow.

Finally, filling the waveform cache is time consuming, requiring several hundred microseconds. Therefore, the PREFETCH instruction allows one to schedule this costly operation during "dead time" in an experiment, e.g. immediately prior to instructions that wait for a trigger.

5.2 Concrete Instructions

The APS2 uses a 64-bit instruction format, divided into header (bits 63-56), and payload (bits 55-0). The format of the payload depends on instruction op code.

5.2.1 Instruction header (8-bits)

Bit(s)	Description
7-4	op code
3-2	engine select (0-3)
1	*reserved*
0	write flag

The op code determines the instruction type. For MARKER instructions, the 'engine select' field chooses the output channel of the instruction. The write flag is used to indicate the final instruction in a group of WAVEFORM and MARKER instructions to be sent simultaneously to their respective execution engines.

5.2.2 Instruction op codes

Code	instruction
0x0	WAVEFORM
0x1	MARKER
0x2	WAIT
0x3	LOAD_REPEAT
0x4	REPEAT
0x5	CMP
0x6	GOTO
0x7	CALL
0x8	RETURN
0x9	SYNC
0xA	MODULATOR
0xB	LOAD_CMP
0xC	PREFETCH

5.2.3 Instruction payload (56-bits)

The 56-bit payload formats for the various instruction op codes are described below.

5.2.4 WAVEFORM

Bit(s)	Description
47-46	op code (0 = play waveform, 1 = wait for trig, 2 = wait for sync, 3 = prefetch)
45	T/A pair flag
44-24	count
23-0	address

The top two bits of the WAVEFORM payload are an op code for the waveform engine. A PLAY_WAVEFORM op code causes the waveform engine to play the waveform starting at *address* for *count* quad-samples. When the time/amplitude pair flag is set, the waveform engine will create a constant- amplitude waveform by holding the analog output at the value given at *address* for *count* quad-samples. The WAIT_FOR_TRIG and WAIT_FOR_SYNC op codes direct the waveform engine to pause until receipt of an input trigger or a sequence SYNC input, respectively. The PREFETCH op code causes the waveform cache to prefetch 64k samples from *addresss* into the pending waveform cache bank.

5.2.5 MARKER

Bit(s)	Description
47-46	op code (0 = play marker, 1 = wait for trig, 2 = wait for sync)
45-37	*reserved*
36-33	transition word
32	state
31-0	count (firmwave versions 2.5-2.33 support only 20 bit count)

The top two bits of the MARKER payload are an op code for the marker engine. A PLAY_MARKER op code causes the marker engine to hold the marker output in value *state* for *count* quad-samples. When the count reaches zero, the marker engine will output the 4-bit transition word. One use of this transition word is to achieve single- sample resolution on a low-to-high or high-to-low transition of the marker output. The WAIT_FOR_TRIG and WAIT_FOR_SYNC op codes function identically to the WAVEFORM op codes.

5.2.6 CMP

Bit(s)	Description
9-8	cmp code (0 = equal, 1 = not equal, 2 = greater than, 3 = less than)
7-0	mask

The CMP operation compares the current value of the 8-bit comparison register to *mask* using the operator given by the *cmp code*. The result of this comparison effects conditional execution of following GOTO, CALL, and RETURN instructions.

5.2.7 LOAD_CMP

Loads the comparison register with the next value from the message queue (received from the trigger distribution module). If the queue is empty, this instruction will halt execution until a message arrives. This instruction ignores all payload data.

5.2.8 GOTO, CALL, and REPEAT

Bit(s)	Description
25-0	address

Jumps to *address*. For GOTO and CALL, the jump may be conditional if proceeded by a CMP instruction. For REPEAT, the jump is conditioned on the repeat counter.

5.2.9 LOAD_REPEAT

Bit(s)	Description
15-0	repeat count

The *repeat count* gives the number of times a section of code should be repeated, i.e. to execute a sequence *N* times, one uses a repeat count of *N-1*.

5.2.10 PREFETCH

Bit(s)	Description
25-0	address

Prefetches a cache-line (128 instructions) starting at *address* into the subroutine cache.

5.2.11 WAIT and SYNC

Bit(s)	Description
47-46	op code (0 = play waveform/marker, 1 = wait for trig, 2 = wait for sync)

The payloads for the WAIT and SYNC instructions must also be valid WAVEFORM and MARKER payloads. Therefore, in addition to indicating WAIT or SYNC in the instruction header, the instruction type must also appear in the payload. The write flag should be set to immediately dispatch this instruction.

5.2.12 RETURN

This instruction ignores all payload data.

5.2.13 MODULATOR

Bit(s)	Description
47-45	op code
44	reserved
43-40	nco select
39-32	reserved
31-0	payload

The modulator op codes are enumerate as follows:

0x0 modulate using selected nco for count (payload)

0x1 reset selected nco phase accumulator

0x2 wait for trigger

0x3 set selected nco phase increment (payload)

0x4 wait for sync

0x5 set selected nco phase offset (payload)

0x6 reserved

0x7 update selected nco frame (payload)

The nco select bit field gives one bit to each NCO. In the current firmware there are two NCO's. For example, to set the frequency of the second NCO the bit field would read `0010` or to reset the phase of both NCOs it would read `0011`.

All phase payloads are fixed point integers UQ2.28 representing portions of a circle. The frequency is determined with respect to the 300MHz system clock. For example, setting a phase increment of 1/3 * 2^28 = 0x02aaaaab gives

a modulation frequency of 50MHz. Integers greater than 2 give frequencies greater than the Nyquist frequency of 600MHz and will be folded back in as negative frequencies.

5.3 Example Sequences

5.3.1 Ramsey

To give a concrete example of construction of a standard QIP experiment in the APS2 format, consider a Ramsey experiment consisting of two $\pi/2$-pulses separated by a variable delay. If the waveform memory has a null-pulse at offset 0x00 and a 16-sample $\pi/2$-pulse at offset 0x01, then the Ramsey sequence might in abstract format would look like:

```
SYNC
WAIT
WAVEFORM 0x01 4
WAVEFORM T/A 0x00 10
WAVEFORM 0x01 4
SYNC
WAIT
WAVEFORM 0x01 4
WAVEFORM T/A 0x00 20
WAVEFORM 0x01 4
SYNC
WAIT
WAVEFORM 0x01 4
WAVEFORM T/A 0x00 30
WAVEFORM 0x01 4
    .
    .
    .
GOTO 0x00
```

The {SYNC, WAIT} sequences demarcate separate Ramsey delay experiments, where the SYNC command ensures that there is no residual data in any execution engine before continuing, and the WAIT command indicates to wait for a trigger. The GOTO command at the end of the sequence is crucial to ensure that the instruction decoder doesn't "fall off" into garbage data at the end of instruction memory.

5.3.2 CPMG

The Carr-Purcell-Meiboom-Gill pulse sequence uses a repeated delay-π-delay sequence to refocus spins in a fluctuating environment. The π pulse is offset by 90 degrees to the intial $\pi/2$ pulse that creates the coherence and even numbers of π pulses are prefered for robustness. We can pull in many elements of arbitrary flow control to compactly describe this sequence. We will use a waveform library with three entries: a null pulse at offset 0x00, a 16-sample $\pi/2$-pulse at offset 0x01, and a 16-sample π-pulse at offset 0x05. Note that offsets are also written in terms of quad-samples, so the memory address range of the first $\pi/2$ pulse is [0x01,0x04]. The Hahn echo delay-π-delay is considered a subroutine. To ensure even multiples a CPMG subroutine then loops over the Hahn echo twice. The two subroutines are placed at the cache-line aligned address 1024 Then a CPMG sequence with 2, 4, 8, 16 . . . loops is:

```
    SYNC
    WAIT
    WAVEFORM 0x01 4 # first 90
    LOAD_REPEAT 0
    CALL 1024 # call the CPMG subroutine
```

(continues on next page)

(continued from previous page)

```
      REPEAT 3
      LOAD_REPEAT 1
      CALL 1024 # call the CPMG subroutine
      REPEAT 6
      LOAD_REPEAT 3
      CALL 1024 # call the CPMG subroutine
      REPEAT 6
      LOAD_REPEAT 7
      CALL 1024 # call the CPMG subroutine
      REPEAT 9
                        .
                        .
                        .
      WAVEFORM 0x01 4 # final 90
      GOTO 0x00
NOOP
      NOOP
      NOOP
                        .
                        .
                        .
      # pad with NOOP's to address 1024
      # start CPMG subroutine
      LOAD_REPEAT 1
      CALL 1028
      REPEAT 1024
      RETURN
      # start Hahn echo subroutine
      WAVEFORM T/A 0x00 25 # delay
      WAVEFORM 0x05 4 # π pulse
      WAVEFORM T/A 0x00 25 #delay
      RETURN
```

5.3.3 Active Qubit Reset

Here we dynamically steer the sequence in response to a qubit measurment in order to actively drive the qubit to the ground state:

```
GOTO 0x06 # jump over 'Reset' method definition
# start of 'Reset' method
WAIT # wait for qubit measurement data to arrive
CMP = 0 # if the qubit is in the ground state, return
RETURN
# otherwise, do a pi pulse
WAVEFORM 0x05 4
GOTO 0x01 # go back to the beginning of 'Reset'
# end of 'Reset' method
SYNC
CALL 0x01 # call 'Reset'
# qubit is reset, do something...
    .
    .
    .
GOTO 0x00
```

In this example, we define a 'Reset' method for flipping the qubit state if it is not currently in the ground state. The

method is defined in instructions 1-5 of the instruction table. We preceed the method definition with a GOTO command to unconditionally jump over the method definition. The structure of the 'Reset' method is a while loop: it only exits when the comparison register is equal to zero. We assume that this register's value is updated to the current qubit state on every input trigger.

CHAPTER 6

Formats

6.1 Waveforms

Stored as arrays of signed 16-bit integers.

6.2 Instructions

Stored as arrays of unsigned 64-bit integers, with the instruction header in the 8 most signficant bits.

6.3 Sequence Files

HDF5 container for waveform and instruction data. The structure of this HDF5 file is as follows:

/version - attribute containing version information for the container structure
/chan_1/instructions - uint64 vector of instruction data
/chan_1/waveforms - int16 vector of waveform data
/chan_2/waveforms - int16 vector of waveform data

CHAPTER 7

API Reference

BBN provides a C-API shared library (libaps2) for communicating with the APS2, as well as MATLAB and Julia wrappers for the driver. We follow language conventions for index arguments, so channel arguments in the C-API are zero- indexed, while in MATLAB and Julia they are one-indexed. Most of the C-API methods require a device serial (an IP address) as the first argument. In MATLAB and Julia, the serial is stored in a device object and helper functions inject it as necessary.

Before calling a device specific API the device must be connected by calling `connect_APS`. This sets up the ethernet interface. Unloading the shared library without disconnecting all APS2s may cause a crash as the library unloading order is uncontrolled. In addition, after every APS2 reset `init_APS` must be called once to properly setup the DAC timing and cache- controller.

7.1 Enums

Nearly all the library calls return an `APS2_STATUS` enum. If there are no errors then this will be `APS2_OK`. Otherwise a more detailed description of the error an be obtained from `get_error_msg`. See the Matlab and Julia drivers for examples of how to wrap each library call with error checking. The enum and descriptions can be found `APS2_errno.h`.

There are also enums for the trigger mode, run mode, running status and logging level. These can be found in `APS2_enums.h` or `logger.h`.

7.2 High-level methods

Getter calls return the value in the memory referenced by the passed pointer. The caller is responsible for allocating and managing the memory.

*const char *get_error_msg(APS2_STATUS)*

Returns the null-terminated error message string associated with the `APS2_STATUS` code.

*APS2_STATUS get_numDevices(unsigned int *numDevices)*

This method sends out a broadcast packet to find all APS2's on the local subnet and returns the number of devices found.

*APS2_STATUS get_device_IPs(const char **deviceIPs)*

Populates *deviceIPs[]* with C strings of APS2 IP addresses. The caller is responsible for sizing deviceIPs appropriately. For example, in C++:

```
int numDevices = get_numDevices();
const char** serialBuffer = new const char*[numDevices];
get_device_IPs(serialBuffer);
```

*APS2_STATUS connect_APS(const char *deviceIP)*

Connects to the APS2 at the given IP address.

*APS2_STATUS disconnect_APS(const char *deviceIP)*

Disconnects the APS2 at the given IP address.

*APS2_STATUS reset(const char *deviceIP, APS2_RESET_MODE)*

Resets the APS2 at the given IP address. The *resetMode* parameter can be used to do a hard reset from non-volatile flash memory to either the user or backup image or can reset the TCP connection should the host computer not cleanly close it.

*APS2_STATUS init_APS(const char *deviceIP, int force)*

This method initializes the APS2 at the given IP address. This involves synchronizing and calibrating the DAC clock timing and setting up the cache-controller. If *force* = 0, the driver will attempt to determine if this procedure has already been run and return immediately. To force the driver to run the initialization procedure, call with *force* = 1.

*APS2_STATUS get_firmware_version(const char *deviceIP, uint32_t *version, uint32_t *git_sha1, uint32_t *build_timestamp, char *version_string)*

Returns computer and humand readable firmware version information. *version* returns the version number of the currently loaded firmware. The major version number is contained in bits 15-8, while the minor version number is in bits 7-0. So, a returned value of 513 indicates version 2.1. Bits 28-16 give the number of commits since the tag and the top nibble set to d indicates a dirty working tree. *git_sha1* is the first 8 hexadecimal digits of the git SHA1 of the latest commit. *build_timestamp* is the build timestamp as a hexadecimal string YYMMDDhh. The *version_string* will combine the previous values into a human readable string similar to what is returned from *git describe*. Pass a null pointer for any unused terms.

*APS2_STATUS get_uptime(const char *deviceIP, double *upTime)*

Returns the APS2 uptime in seconds.

*APS2_STATUS set_sampleRate(const char *deviceIP, unsigned int rate)*

Sets the output sampling rate of the APS2 to *rate* (in MHz). By default the APS2 initializes with a rate of 1200 MHz. The allow values for rate are: 1200, 600, 300, and 200. **WARNING**: the APS2 firmware has not been tested with sampling rates other than the default of 1200. In particular, it is expected that DAC synchronization will fail at other update rates.

*APS2_STATUS get_sampleRate(const char *deviceIP, unsigned int *rate)*

Returns the current APS2 sampling rate in MHz.

*APS2_STATUS set_channel_offset(const char *deviceIP, int channel, float offset)*

Sets the offset of *channel* to *offset*. Note that the APS2 offsets the channels by digitally shifting the waveform values, so non-zero values of offset may cause clipping to occur.

*APS2_STATUS get_channel_offset(const char *deviceIP, int channel, float *offset)*

Returns the current offset value of *channel*.

*APS2_STATUS set_channel_scale(const char *deviceIP, int channel, float scale)*

Sets the scale parameter for *channel* to *scale*. This method will cause the currently loaded waveforms (and all subsequently loaded ones) to be multiplied by *scale*. Values greater than 1 may cause clipping.

*APS2_STATUS get_channel_scale(const char *deviceIP, int channel, float *scale)*

Returns the scale parameter for *channel*.

*APS2_STATUS set_channel_enabled(const char *deviceIP, int channel, int enabled)*

Enables (*enabled* = 1) or disables (*enabled* = 0) *channel*. **Currently non-functional**

*APS2_STATUS get_channel_enabled(const char *deviceIP, int channel, int *enabled)*

Returns the enabled state of *channel*.

*APS2_STATUS set_mixer_amplitude_imbalance(const char * deviceIP, float amp)*

Set the mixer amplitude imbalance tp *amp* and updates the correction matrix.

*APS2_STATUS get_mixer_amplitude_imbalance(const char * deviceIP, float *amp)*

Gets the mixer amplitude imbalance.

*APS2_STATUS set_mixer_phase_skew(const char * deviceIP, float skew)*

Sets the mixer phase skew (radians) to *skew* and updates the correction matrix.

*APS2_STATUS get_mixer_phase_skew(const char * deviceIP, float *skew)*

Gets the mixer phase skew (radians).

*APS2_STATUS set_mixer_correction_matrix(const char * deviceIP, float *matrix)*

Sets the complete 2x2 mixer correction matrix. Pass an array of four float to fill the matrix in row major order.

*APS2_STATUS get_mixer_correction_matrix(const char * deviceIP, float *matrix)*

Gets the complete 2x2 mixer correction matrix in row major order.

*APS2_STATUS set_trigger_source(const char *deviceIP, APS2_TRIGGER_SOURCE source)*

Sets the trigger source to EXTERNAL, INTERNAL, SYSTEM, or SOFTWARE.

*APS2_STATUS get_trigger_source(const char *deviceIP, APS2_TRIGGER_SOURCE *source)*

Returns the current trigger source.

*APS2_STATUS set_trigger_interval(const char *deviceIP, double interval)*

Set the internal trigger interval to *interval* (in seconds). The internal trigger has a resolution of 3.333 ns and a minimum interval of 6.67ns and maximum interval of `2^32+1 * 3.333 ns = 14.17s`.

*APS2_STATUS get_trigger_interval(const char *deviceIP, double *interval)*

Returns the current internal trigger interval.

*APS2_STATUS trigger(const char *deviceIP)*

Sends a software trigger to the APS2.

*APS2_STATUS set_waveform_float(const char *deviceIP, int channel, float *data, int numPts)*

Uploads *data* to *channel*'s waveform memory. *numPts* indicates the length of the *data* array. ± 1 indicate full-scale output.

*APS2_STATUS set_waveform_int(const char *deviceIP, int channel, int16_t *data, int numPts)*

Uploads *data* to *channel*'s waveform memory. *numPts* indicates the length of the *data* array. Data should contain 14-bit waveform data placed into the lower 14 bits (13-0) of each int16 element. Bits 15-14 in each array element will be ignored.

*APS2_STATUS set_markers(const char *deviceIP, int channel, uint8_t *data, int numPts)*

FOR FUTURE USE ONLY Will add marker data in *data* to the currently loaded waveform on *channel*.

*APS2_STATUS write_sequence(const char *deviceIP, uint64_t *data, uint32_t numWords)*

Writes instruction sequence in *data* of length *numWords*.

*APS2_STATUS load_sequence_file(const char *deviceIP, const char* seqFile)*

Loads the APS2-structured HDF5 file given by the path *seqFile*. Be aware the backslash character must be escaped (doubled) in C strings.

*APS2_STATUS set_run_mode(const char *deviceIP, APS2_RUN_MODE mode)*

Changes the APS2 run mode to sequence (RUN_SEQUENCE, the default), triggered waveform (TRIG_WAVEFORM) or continuous loop waveform (CW_WAVEFORM) **IMPORTANT NOTE** The run mode is not a state and the APS2 does not "remember" its current playback mode. The waveform modes simply load a simple sequence to play a single waveform. In particular, uploading new sequence or waveform data will cause the APS2 to return to 'sequence' mode. To use 'waveform' mode, call *set_run_mode* only after calling *set_waveform_float* or *set_waveform_int*.

*APS2_STATUS set_waveform_frequency(const char *deviceIP, float freq)*

Sets the modulation frequency for waveform run mode to *freq*.

*APS2_STATUS get_waveform_frequency(const char *deviceIP, float *freq)*

Gets the modulation frequency for waveform run mode.

*APS2_STATUS run(const char *deviceIP)*

Enables the pulse sequencer.

*APS2_STATUS stop(const char *deviceIP)*

Disables the pulse sequencer.

*APS2_STATUS get_runState(const char *deviceIP, APS2_RUN_STATE *state)*

Returns the running state of the APS2.

*APS2_STATUS get_mac_addr(const char *deviceIP, uint64_t *MAC)*

Returns the MAC address of the APS2 at the given IP address.

*APS2_STATUS set_ip_addr(const char *deviceIP, const char *ip_addr)*

Sets the IP address of the APS2 currently at *deviceIP* to *ip_addr*. The IP address does not actually update until *reset()* is called, or the device is power cycled. Note that if you change the IP and reset you will have to disconnect and re-enumerate for the driver to pick up the new IP address.

7.3 Low-level methods

int set_log(char logfile)*

Directs logging information to *logfile*, which can be either a full file path, or one of the special strings "stdout" or "stderr".

int set_logging_level(TLogLevel level)

Sets the logging level to *level* (values between 0-8 logINFO to logDEBUG4). Determines the amount of information written to the APS2 log file. The default logging level is 2 or logINFO.

*int write_memory(const char *deviceIP, uint32_t addr, uint32_t* data, uint32_t numWords)*

Write *numWords* of *data* to the APS2 memory starting at *addr*.

*int read_memory(const char *deviceIP, uint32_t addr, uint32_t* data, uint32_t numWords)*

Read *numWords* into *data* from the APS2 memory starting at *addr*.

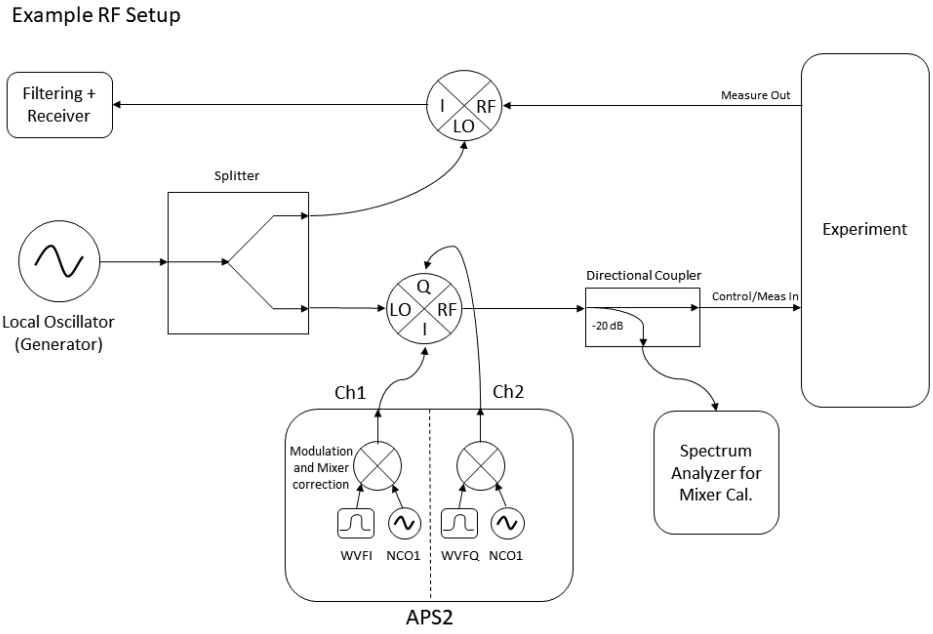
*int read_register(const char *deviceIP, uint32_t addr)*

Returns the value of the APS2 register at *addr*.

CHAPTER 8

Experiment Setup

The Arbitrary Pulse Sequencer 2 (APS2) modules operate with 1.2 GS/s DACs. Each card supports two output channels, typically used as the in-phase and quadrature for a single qubit. Qubit measurement and control frequencies are higher than the frequencies achievable from the output of the APS2 (does not support higher Nyquist zones). Consequently, it is necessary to mix the APS2 output with an RF local oscillator. A diagram of a typical experiment set up with an APS2 is provided below:



Examples in this documentation reference BBN's QGL v1.1.x (http://github.com/BBN-Q/QGL) and Auspex v0.4.x (https://github.com/BBN-Q/Auspex).

Further detail about the use of an APS2 in a qubit experiment can be found here: https://aip.scitation.org/doi/full/10.1063/1.5006525

Example .yml configuration files and an IPython notebook (*Single_Qubit_Characterization.ipynb*) can be found in the `examples` section of libaps2 (http://github.com/BBN-Q/libaps2).

8.1 YAML Setup

8.1.1 Measure.yml

For an example declaration of an APS2 in the Channel library, look at *measure.yml*. Be sure to update the relevant directory paths in the file. In the example, we associate qubit Q1 with BBNAPS1 12 (measure) and BBNAPS2 12 (control). The `12` after the BBNAPS identifier specifies the use of channel 1 for in-phase control and channel 2 for quadrature control. For measurement channels, one should provide a `trigger` output to pass to the digitizer card. This trigger will indicate when a measurement has begun so data collection can start. Each APS2 includes a `generator` in the library. The generator is the Local Oscillator associated with the measure or control channel. The `autodyne_freq` is the frequency with which the APS2 modulates each pulse. The APS2 can support up to two onboard NCO's up to 600 MHz each (typically utilized at <200 MHz). The added feature of multiple NCO's allows one APS2 card to drive *1:* I/Q modulation on two measurement frequencies (multiplexed readout) or *2:* both the single qubit and two-qubit control pulses. During qubit and cavity spectroscopy, the LO frequency is typically swept and the `autodyne_freq` is kept at zero. Once, the cavity and qubit frequency are settled upon, a modulation (10's of MHz) is added to the APS2 such that $f_{m,c} = f_{LO} + f_{mod}$. $f_{m,c}$ is the targeted measure or control frequency and f_{mod} is the added frequency calculated within the APS2. By utilizing a non-zero modulation, LO leakage is not directly resonant with the feature of interest. The `pulse_params` are the actual parameters of the measure and control pulses. The pulses are scaled by the factors in the instruments.yml file to account for physical imperfections. A pulse from the APS2 will have a rising/falling edge given by the `shape_fun` parameter and run for a total duration given by the `length` parameter. Certain shapes require additional terms, for example the `tanh` function requires a `sigma` value which defines the rise time. Source code for these pulse shapes is in *QGL/QGL/PulseShapes.py*. For measure pulses, there is only one overall amplitude scale factor. For control pulses the user can specify and amplitude for $\pi/2$ and π pulses (`pi2amp` and `piamp`). Auspex calibration routines (*cal.Pi2Calibration* and *cal.PiCalibration*) help precisely set these control parameters. The `cutoff` parameter should be set to 2.0. And finally, the `gain` parameter is wholly a dummy term. It is meant to help users track the added physical attenuation in line with the output. `Markers` which will be used in QGL are nicknamed in the measure.yml file. Markers are the four digital output channels from the APS2. In this case, we have defined `mark_M3` which corresponds to the third physical marker on BBNAPS2. Each physical marker should be defined in instruments.yml.

8.1.2 Instruments.yml

8.1.3 APS2

All physical instruments are described in *instruments.yml*. The two APS2s referenced in the measure.yml file are declared here. Each APS2 is associated with its IP address, identified as a master or slave, and assigned various pulse parameters. One should also declare physical `markers`. If a TDM is in use as the system trigger, it also enumerated. Further detail on APS2 triggering is provided in the next section. For an APS2, the `tx_channels` field is where one specifies the output amplitude, phase and offset for the APS2. The actual amplitude emitted is the product the amplitude in the qubit declaration (measure.yml), and the `amplitude` field for each output channel. Anecdotally, the APS2 output power is linear (doubling the amplitude results in 3 dB output power increase) below a total scale factor of 0.7. The `offset`, `amp_factor`, and `phase_skew` fields are related to mixer imperfections. These three parameters are typically tuned using Mixer Calibration routines, combined with the Spectrum Analyzer measurements depicted in the figure above. Further discussion of mixer calibration is provided below.

8.1.4 Markers

There are four markers associated with each APS2 (typically 12m1-4). They can be activated in two ways. The first way is through the YAML files. In the example measure.yml provided, for BBNAPS1, marker 2 is declared as the measurement trigger. The properties in the instruments.yml dictate that it will wait 150 ns after the measure pulse begins and emit a pulse of 200 ns. The second way that markers are activated is through the QGL command *TRIG*. Given the marker declared above, utilizing QGL, one could declare *m3 = MarkerFactory("mark_M3")*. Then the command *TRIG(m3,100e-9)*, would activate the marker for 100 ns within a QGL gate sequence. This utilization of markers can be a tool in debugging the APS2 output on an oscilloscope.

8.1.5 Generators

Generators (LO's) are also enumerated in the instruments.yml file (see `Microwave Sources`). In the example, the measurement of Q1 is associated with `Autodyne M1` which is a Vaunix Labbrick signal generator. The LO for control pulses on Q1 is created with `q1source`. This is physically generated via an AgilentN5183A. A commented-out template for a Holzworth RF source is also provided (`AutodyneM2`). Generators are typically powered in the region specified by the IQ mixer (e.g. ~13 dBm for Marki Microwave IQ-0307). Each generator should be locked to an external `reference`, usually 10 MHz, to perform repeatable phase coherent experiments.

8.2 Triggering

Following the trigger documentation: 'The APS2 supports four different types of triggers. The internal mode generates triggers on a programmable interval between 6.66ns and 14s. The external mode listens for triggers on the front-panel SMA "trigger input" port. In this mode, the APS2 is triggered on the rising edge of a 1-5V signal. The system trigger accepts triggers on the SATA input port from the APS2 Trigger Distribution Module (TDM). Finally, the software mode allows the user to trigger the APS2 via the host computer with the trigger() API method'. Here we will focus on two modes: system triggering and master/slave triggering.

8.2.1 System

To implement system triggering, set all APS2 `trigger_source` to `system` (and `master` to false). A TDM must be enabled, with its `trigger_source` set to Internal. In this way, the TDM will generate a trigger pulse every `trigger_interval` seconds. This pulse is distributed to the APS2 units via the SATA interconnects, at which point the APS2 sequencers begin. This method is preferable if more than few APS2 units are in use. The SATA distribution system is far easier than connecting a single APS2 master marker to many slaves.

8.2.2 Master/Slave

Instead, if the APS2 is to act as the `master`, one should specify the `slave_trig` marker. The master should be set to an Internal `trigger_source`. All the slaves should have an External `trigger_source`. The master will then generate a pulse every `trigger_interval` seconds on its `slave_trig` line. This marker should be plugged into all the slave APS2 units on their trigger input port. Note, the output of the marker cannot support more than a few APS2 units without amplification. In all cases the `trigger_interval` on the trigger generator should be much longer than an experiment, to allow for qubit relaxation.

8.3 Mixer Calibration Routines

There are imperfections that can result from the mix-up process with an LO. In particular, there is leakage at the LO frequency (assuming non-zero I/Q frequency), gain imbalance in the I and Q channels, and phase imbalance in the I and Q channels. We seek to correct these with the mixer calibration routines: *QubitExpFactory.calibrate_mixer*, found in *Auspex mixer_calibration.py*. The correction schemes are based upon Analog Devices AN-1039. The routines utilize a BBN Spectrum analyzer (SA in intstruments.yml) which requires its own local oscillator (SALO). By tuning the DC offset on the mixer, we can suppress leakage at the LO frequency. Similarly, by adjusting a constant phase (`phase_skew`) to all waveforms, we can correct for I/Q phase imbalance in the mixer. And by rescaling the I and Q relative amplitudes, we can correct for a gain mismatch in the two sides of the IQ mixer. Sweeps of these parameters are performed and the returned data from the spectrum analyzer is fitted to determine optimal parameters. The scripts will then update both of the channel offsets, phase skew and amplitude factor in instruments.yml. The APS2 is responsible for performing the correct on-board 2x2 matrix multiplication and offset addition to apply these fixes.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search