Synchronizing USRP Events Using Timed Commands in UHD

Contents

AN-883

This AN discusses Timed Commands in UHD. We will explore use cases, theory of operation, and multi_usrp based examples of timed command used in UHD 3.x.

Timed Commands are an important aspect of using a USRP. They allow a user to coordinate USRP state changes with nanosecond precision across multiple devices. Examples of timed command use cases include:

- Configuring multiple channels of a single USRP to change frequency simultaneously.
- Configuring the channels on multiple USRPs to synchronously retune and ensure a predictable phase offset between channels (if supported by daughterboard).
- Changing the state of a USRP's GPIO line at an absolute time.
- Coordinating a simultaneous change of gain and frequency in a USRP.
- Scheduling frequency hopping at set time increments.
- Using RFNoC blocks like the Replay Block to transmit phase coherent bursts.

A common use case of timed commands is to ensure a predictable and repeatable phase offset between the channels of multiple USRPs. Before we dive into the low level details of timed commands, lets take a high level look at where they are used in configuring a phase coherent MIMO system.

There are four key elements required for phase coherent operation of resync-capable USRPs:

1. All USRPs share a common reference clock (10MHz Ref)
2. All USRPs share a common sense of time (PPS)
3. LO and DSP tuning is synchronous
4. Streaming is started synchronously

LO sharing is implemented in some USRP products, but will not be covered in this example. Here is a physical configuration satisfying the requirements listed above:

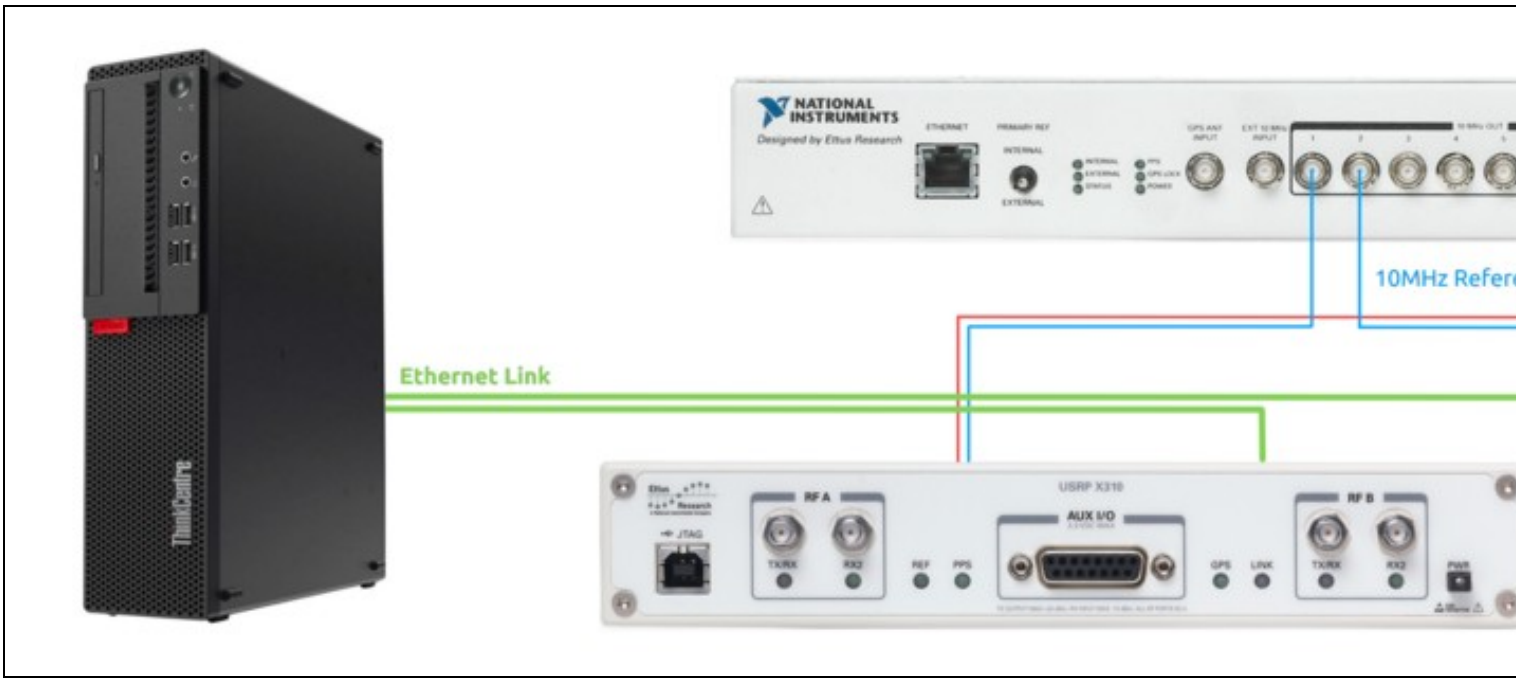

Figure 1 - Example system configuration with Host PC, Octoclock, and X310 radios

Now that the system has all the physical connections necessary, we need to coordinate things in our host code. This will include importing the 10MHz Ref and PPS as well as ensuring synchronous tuning and streaming.

First, we need to create a `multi_usrp` object that contains both USRPs in our physical configuration:

```
uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::make("addr0=192.168.10.2,addr1=192.168.10.3");
```

Next, we'll let the USRPs know that they should expect a 10MHz clock on their "Ref In" port, and a PPS signal on their "PPS In" port:

```
usrp->set_clock_source("external");
usrp->set_time_source("external");
```

At this point, both USRPs are locked to an external reference and a common PPS signal. Next we want to tell both USRPs to reset their sense of time to 0.000s on the next PPS edge:

```
usrp->set_time_next_pps(uhd::time_spec_t(0.0));
std::this_thread::sleep_for(std::chrono::milliseconds(1000));
```

We won't cover it much in this application note, but for now understand `time_spec_t` is the UHD API's means of formatting time values for a USRP. After we reset the USRP's sense of time, we wait 1 second to ensure a PPS rising edge occurs and latches the 0.000s value to both USRPs. At this point, both USRPs should have a shared sense of time. We've now satisfied the first and second requirements for phase coherent USRP operation. Let's move on to synchronously tuning using timed commands:

```
usrp->clear_command_time();

usrp->set_command_time(usrp->get_time_now() + uhd::time_spec_t(0.1)); //set cmd time for .1s in the future

uhd::tune_request_t tune_request(freq);
usrp->set_rx_freq(tune_request);
std::this_thread::sleep_for(std::chrono::milliseconds(110)); //sleep 110ms (~10ms after retune occurs) to allow LO to lock

usrp->clear_command_time();
```

With the above code block, we are able to set a command time equal to the current time + 0.1s. Any commands that are called after `set_command_time()` will be sent to the USRP with a timestamp corresponding to the argument passed to `set_command_time()`. Because of this timestamp, the USRP will wait until the command time passes to execute the tune request. This will ensure that the LO and DSP chain of our USRPs are retuned synchronously (on the same clock cycle), satisfying the third requirement for phase coherent operation.

The final step in configuring our system is to set up synchronous streaming from both radios. For this example, we'll set up synchronous RX streaming using the following code:

```
// create a receive streamer
uhd::stream_args_t stream_args("fc32", wire); // complex floats
stream_args.channels             = "0,1,2,3";
uhd::rx_streamer::sptr rx_stream = usrp->get_rx_stream(stream_args);

// setup streaming
uhd::stream_cmd_t stream_cmd(uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS);
stream_cmd.stream_now = false;
stream_cmd.time_spec  = uhd::time_spec_t(usrp->get_time_now() + uhd::time_spec_t(1.0));
rx_stream->issue_stream_cmd(stream_cmd);
```

Our system will begin to stream data 1s after the time returned by `usrp->get_time_now()`. Data sent to the host will be phase coherent between the 4 RX channels in this system. This architecture allows a system with X310 + UBX daughterboards to maintain a known phase relationship between channels across runs of code and system power cycles.

For more detail, see USRP Manual: Device Synchronization

In this section, we will cover several key topics relating to USRP synchronization and the use of timed commands in UHD. This is the low level detail section of this application note.

PPS is a signal used by USRPs for time synchronization. With a shared PPS, the sense of time can be aligned across several USRPs, allowing for the synchronization of timed command execution on an arbitrary number of radio channels. Within the context of a USRP, a PPS signal is expected to have the following properties:

- 1Hz
- TTL Signal Levels
- 25% duty cycle

A USRP's PPS can be derived from a GPSDO automatically, from an externally supplied PPS signal, or via internal PPS synthesis (not supported in legacy USRPs).

A PPS trigger is used to coordinate time alignment events across multiple devices. For example, the USRPs internal sense of time (the `vita_time` counter) can be synchronously set/reset across multiple USRPs via UHD API calls such as:

```
set_time_next_pps
```

```
set_time_unknown_pps
```

Ettus Research recommends the Octoclock for distribution of PPS and 10MHz REF signals across multiple devices, and the Octoclock-G for this functionality as well as tight alignment with GPS time. For further information on PPS and other common reference signals, see the UHD Manual: Device Synchronization.

CHDR (pronounced "Cheddar", like the cheese) or "Compressed Header" packets are the packet type used to pass data and commands between a host computer and USRP. CHDR is a derivative of the VITA 49 (VRT) protocol which is proprietary to USRP devices. See UHD Manual: Radio Transport Protocols for more information.

There are 4 types of packets used in the USRP:

- Data
- Flow Control
- Command
- Command Response

The type of CHDR packet is determined by the value of bits 63 and 62 in the CHDR header:

CHDR Packet Form

Address (Bytes)	Length (Bytes)	Payload
0	8	Compressed Header (CHDR)
8	8	Fractional Time (Optional!)
8/16	-	Data

Compressed Header Bit Meaning

Bits	Meaning
63:62	Packet Type
61	Has fractional time stamp (1: Yes)
60	End-of-burst or error flag
59:48	12-bit sequence number
47:32	Total packet length in Bytes
31:0	Stream ID (SID). For the format of SID, see

Figure 2 - Expanded view of CHDR packet composition

All of these packet types have a single bit (61) used to denote whether an optional timestamp is included. If present, a timestamp is a 64-bit value representing an absolute time value. In this application note, we?re concerned with the use and functionality of command packets with a timestamp present, also known as ?timed commands?.

As commands are passed from host to USRP, they are added to a FIFO on the USRP's FPGA. This FIFO is called the command queue and all commands that are sent to the USRP must pass through a command queue. Each block in the FPGA that handles data also has its own command queue. The command queue FIFO is not to be confused with the data FIFOs used to buffer data between blocks (pictured in Figure 3).

Every command queue maintains a sense of time. The mechanism for acquiring this sense of time is different between the Radio Core and other IP cores (including custom RFNoC blocks) and will be explored later in this application note. When commands enter the command queue, their timestamp is compared against the command queue's sense of time and the commands are executed when Queue Time >= Command Time. Commands without timestamps are executed immediately when they're at the front of the queue. Command queues in the USRP do not support on-the-fly reordering, meaning a command at the front of the queue will block subsequent commands from executing even if their timestamp has passed.

Every IP core on a USRP, including the Radio Core, DDC, DUC, and custom blocks, includes one command queue per data stream (certain blocks are designed to pass multiple data streams). The depth of this command queue varies from device to device is determined at FPGA compilation time based on user settings and available resources. An overflow of the command queue will result in a system halt and often requires a physical reset of the FPGA. Here are the default command queue depths for various USRP models:

USRP	FPGA	Default CMD Queue Depth (Radio Core)	Default CMD Queue Depth (IP Cores)
X300	Xilinx Kintex-7 XC7K325T	8	5
X310	Xilinx Kintex-7 XC7K410T	8	5
E310/E312/E313	Xilinx Zynq 7020 SoC	8	5
E320	Xilinx Zynq 7045 SoC	8	5
N300	Xilinx Zynq-7035 SoC	8	5

N310/N320/N321	Xilinx Zynq-7100 SoC	8	5
B200/B200mini	Xilinx Spartan 6 XC6SLX75	8	5
B210/B205mini	Xilinx Spartan 6 XC6SLX150	8	5
N200	Xilinx Spartan 3A-DSP XC3SD1800A	64	64
N210	Xilinx Spartan 3A-DSP XC3SD3400A	64	64

Table 1 - Default command queue depth of various USRPs

The Radio Core is the heart of the USRP's functionality. The radio core is responsible for controlling all TX and RX daughterboard components (synthesizer, signal path, gain and attenuation elements, etc.), GPIO, setting up data streaming to/from DACs and ADCs, and related error handling.

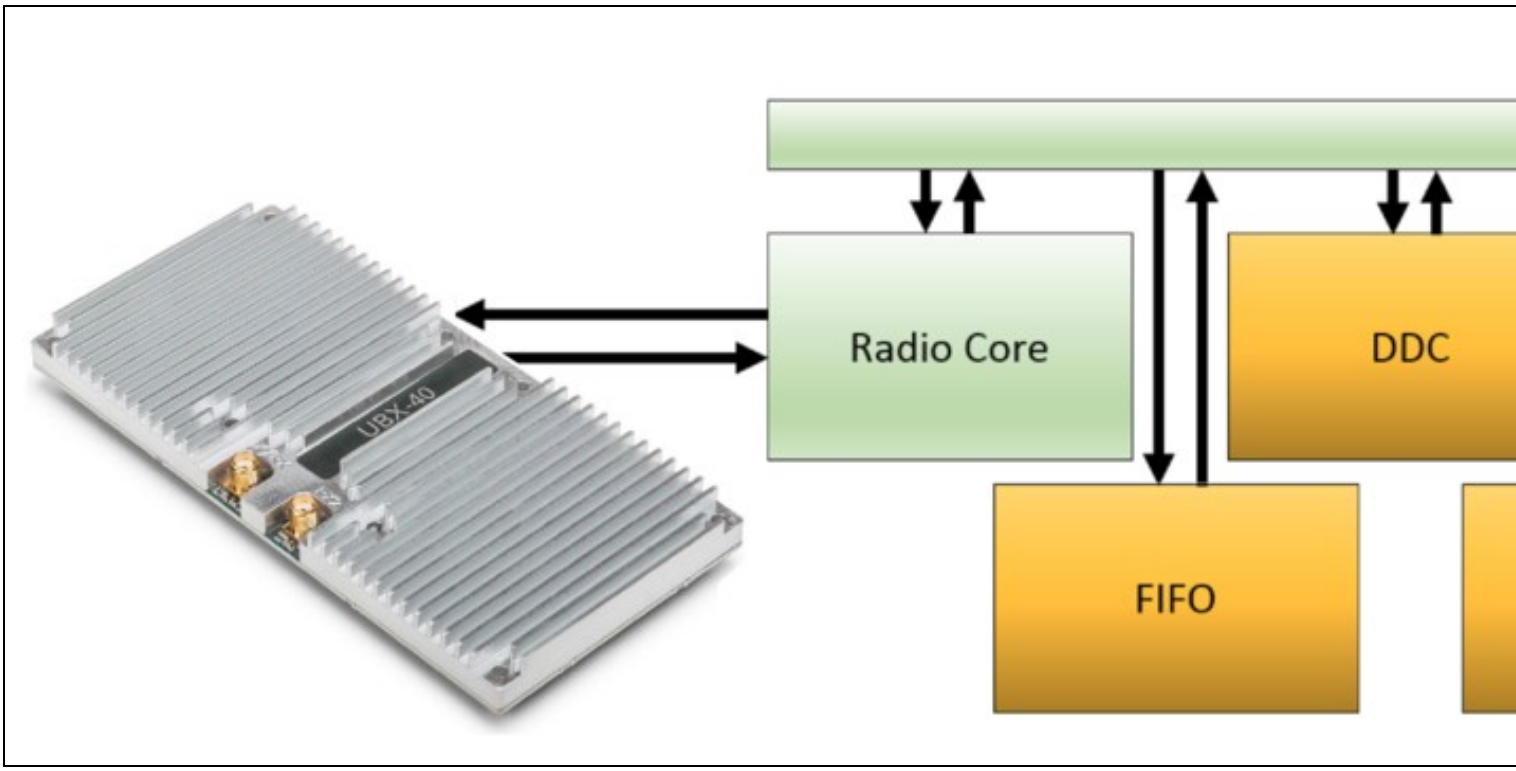

Figure 3 - RFNoC Architecture with UBX daughterboard

In addition to the functionality listed above, the Radio Core is also responsible for maintaining the USRP's sense of absolute time. This absolute time value is stored as a 64-bit counter called the "VITA time" or `vita_time` in UHD. The `vita_time` counter is incremented off of the FPGA's base clock and local to the Radio Core USRP, meaning that other RFNoC blocks can not reference this counter value directly. This sense of absolute time is a subtle yet important difference between the Radio Core and other RFNoC blocks, which can also execute timed commands but do so based on the timestamps (and sample rate) of packets which pass through these blocks.

The next case to cover is the handling of timed commands within FPGA blocks that are not the Radio Core. The Digital Down Converter (DDC) and Digital Up Converter (DUC) are two default USRP FPGA blocks that fall into this category. These blocks are responsible for performing digital frequency shifts on IQ samples that are passed through them. Precise execution of these frequency shifts are essential to phase coherent operation of the USRP.

IP Cores like the DDC and DUC are reliant on the timestamp of packets (and the sample rate of the radio) to derive a sense of time. In the case of the receive chain, samples that are digitized by an ADC are then packaged into a CHDR Packet by the Radio Core (with an included timestamp equal to `vita_time`) and are then passed downstream to the DDC. The DDC will read the timestamps of the incoming samples and apply any queued frequency shift precisely on the first sample with a timestamp >= the command time.

With this method, a USRP can begin receiving data at absolute time `t0`, tune it's LO at an absolute time of `t1`, and then apply a frequency shift in the DDC at the point in data corresponding to `t1`, resulting in a physical and digital frequency change that begin on the exact same sample.

The reverse process holds true for the transmit chain. One often overlooked difference is that the host must pass along at least 1 sample with a timestamp included in the metadata. Without this timestamped packet, the DUC can not derive a sense of time and therefore will never execute timed commands that are in its queue. This will either result in a command queue overflow or a "No Response Packet" runtime error from `ctrl_iface.cpp`.

- The majority timing operations amount a sequence of peeks and pokes to FPGA registers. Examples include gain changes, tune requests, etc. The timestamp of a timed command corresponds to the beginning of this peek/poke sequence.
- The timing of data transmission (TX) is a bit more involved, requiring queueing of samples, and arming of the device for a timed transmission. This is handled by the FPGA when a timestamp is present in the first TX sample's metadata.
- A timed receive operation (RX) requires that a stream command with a timestamp be issued to the radio (see `stream_cmd.time_spec` in the "System Level Example"). This syntax is different than a TX operation and does not involve sample metadata.

- Timed commands are supported on AD93xx-based USRPs (E3xx, B2xx, N300/N310), with a few caveats:
  - Timed commands do not allow for phase resync of the AD93xx internal LO.
  - All functions that directly interact with the AD93xx (tuning, gain change, etc) are subject to the scheduling of the AD93xx. The determinism of these operations are not guaranteed.
- Timed commands for TX and RX on AD936x devices are supported, with caveats:
  - There will be a delay between an absolute time passing and the AD936x actually beginning a streaming operation. This delay has been observed to be consistent *for streaming*, but other operations like gain setting require SPI readback from the RFIC and are non-deterministic.

    - This is to say that a timed streaming command to begin TX and RX at the same time, on an AD936x-based device (which is in external loopback) should result in a consistent delay between TX and RX down to the sample. Other RFIC operations like changing gain or tuning LO frequency will not have this consistency.

- No. Timed commands are passed from host to USRP and are added to a queue while host code progresses.

Modern USRPs pass packets using the CHDR protocol. These packets can be used to issue commands to the USRP and may have an associated timestamp. A command with an included timestamp is called a timed command and it's important to understand how the USRP handles these timed commands. All blocks in a USRP's FPGA have a command queue and maintain a sense of time, however the Radio Core has the unique ability to store an absolute sense of time known as as the `vita_time`. When timed commands are issued to the USRP, they are added to a command FIFO of finite depth and are executed when the timestamp in the header of the command packet is >= the RFNoC block's sense of time. Utilizing these timed commands correctly allows for various USRP functionality to be executed with nanosecond precision, enabling time and phase coherent operation across multiple devices.

In this section, we will list and briefly describe commands that are commonly used in USRP event timing. Some less common methods which require a more detailed explanation can be found in The multi_usrp Class Reference.

Get the current time in the USRP time registers:

```
get_time_now()
```

Get the time when the last pps pulse occurred:

```
get_time_last_pps()
```

Get the currently set time source:

```
get_time_source()
```

Sets the time registers on the USRP immediately:

```
set_time_now()
```

Set the time registers on the usrp at the next pps tick:

```
set_time_next_pps()
```

Set the time source for the USRP device:

```
set_time_source()
```

Clear the command time so future commands are sent ASAP.

```
clear_command_time()
```

Set the time at which the control commands will take effect.

```
set_command_time()
```

Default TX metadata constructor:

```
tx_metadata_t()
```

Indicates whether a TX packet has a timestamp included:

```
md.has_time_spec
```

Sets the timestamp that will be added to a TX packet

```
md.time_spec
```

Default stream command constructor:

```
stream_cmd_t()
```

Member function indicating whether the stream will begin immediately:

```
stream_cmd.stream_now
```

Indicates the point in time for an RX stream to begin:

```
stream_cmd.time_spec
```

Issue a stream command to the USRP:

```
issue_stream_cmd(stream_cmd)
```

The following code uses timed commands to vary a USRP's bottom 4 GPIO lines between high and low. This can be copy-pasted into existing USRP examples:

```
    /////////////////////////////////////////
    /////////// begin gpio operations //////////
    /////////////////////////////////////////

    // set up some catch-all masks
```

```
uint32_t gpio_line = 0xF; // only the bottom 4 lines: 0xF = 00001111 = Pin 0, 1, 2, 3
uint32_t all_one = 0xFF;
uint32_t all_zero = 0x00;

// reset usrp time to 0.00
usrp->set_time_source("internal");
usrp->set_time_next_pps(uhd::time_spec_t(0.0));
std::this_thread::sleep_for(std::chrono::milliseconds(2000));

uhd::time_spec_t now_time = usrp->get_time_last_pps();        // define t=0


// set gpio pins up for output
usrp->set_gpio_attr("FP0", "DDR", all_one, gpio_line, 0);
usrp->set_gpio_attr("FP0", "CTRL", all_zero, gpio_line, 0);
usrp->set_gpio_attr("FP0", "OUT", all_one, gpio_line, 0);     // reset HIGH (async)

// set all gpio lines to output 0
usrp->clear_command_time();
usrp->set_command_time(now_time + uhd::time_spec_t(2.0));
usrp->set_gpio_attr("FP0", "OUT", all_zero, gpio_line, 0);    // set LOW @ t=2

// set all gpio lines to output 1
usrp->clear_command_time();
usrp->set_command_time(now_time + uhd::time_spec_t(4.0));
usrp->set_gpio_attr("FP0", "OUT", all_one, gpio_line, 0);    // set HIGH @ t=4

// set all gpio lines to output 0
usrp->clear_command_time();
usrp->set_command_time(now_time + uhd::time_spec_t(6.0));
usrp->set_gpio_attr("FP0", "OUT", all_zero, gpio_line, 0);   // set LOW @ t=6

usrp->clear_command_time();

/*******************************************/
/*********** quit gpio operations **********/
/*******************************************/
```

A low-cost logic analyzer can be used to monitor the GPIO line states:

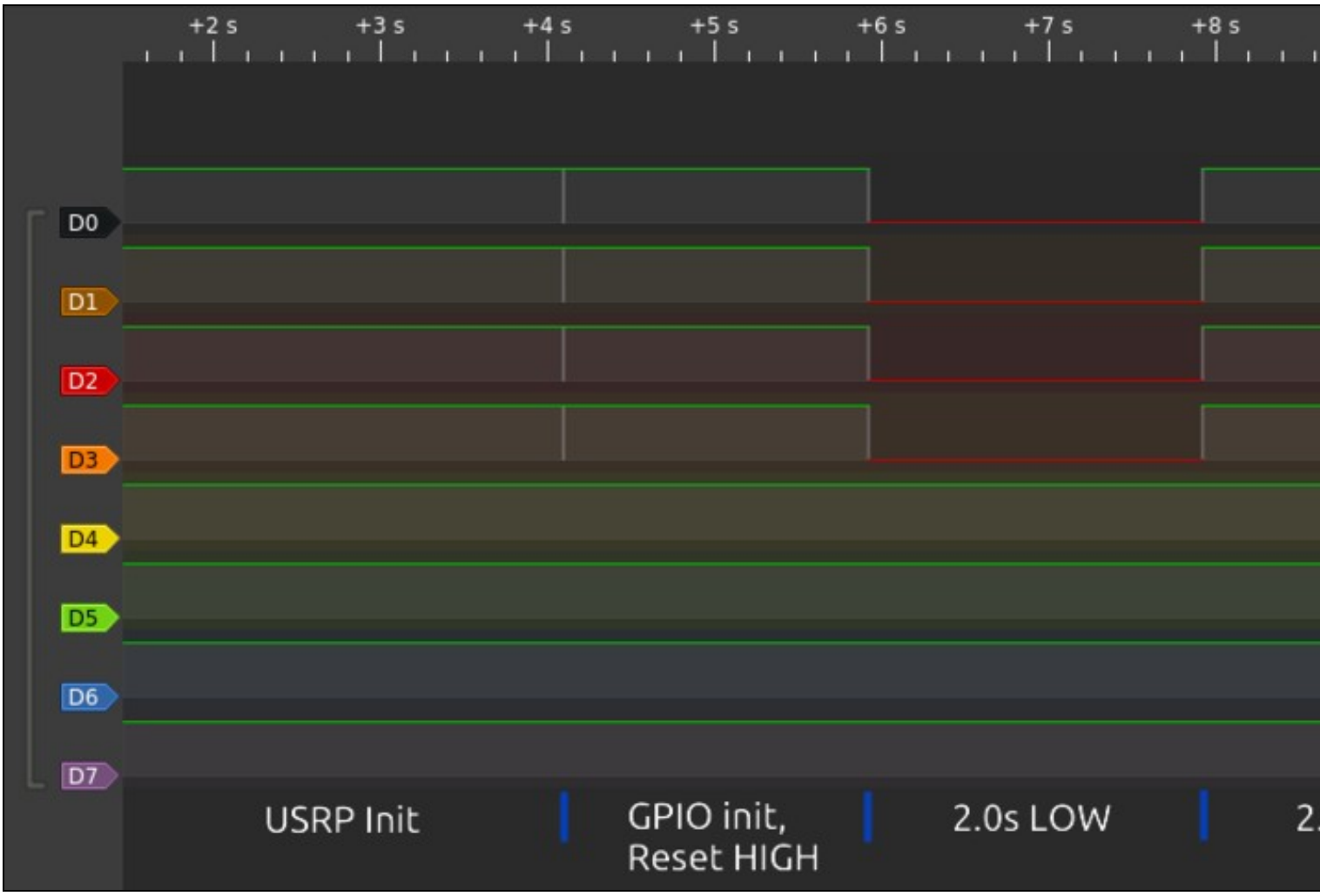

Figure 4 - Logic analyzer readout from timed GPIO example