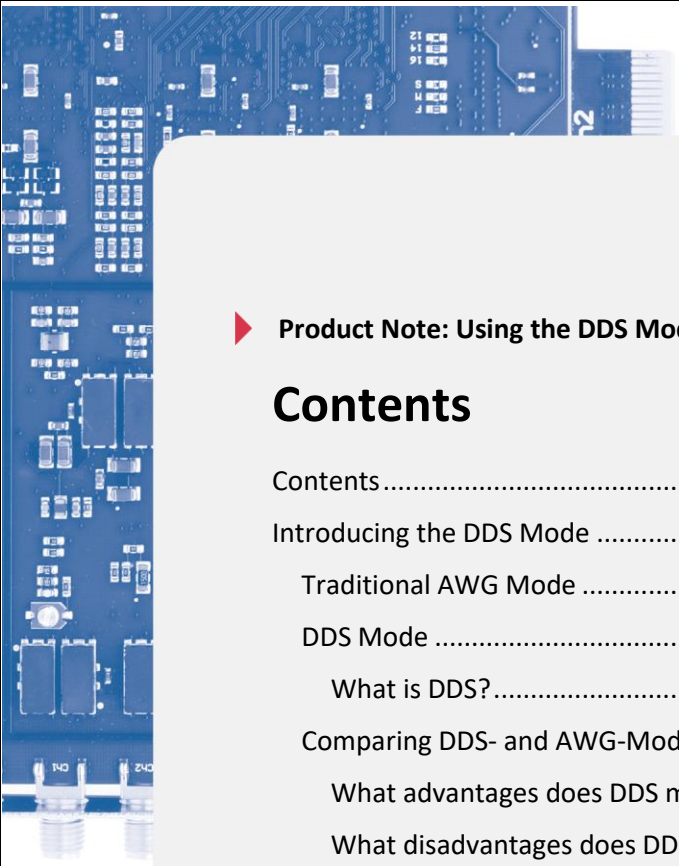
Product Note
Using the DDS Mode

This product note explains using the DDS mode of Spectrum Instrumentation AWGs in detail. The product note explains the usage in a generic way and can be used for any Spectrum Instrumentation AWG product that has a DDS mode implemented or available as an option. All examples are done for the M4i.66xx AWG series but will also work with other models.



SPECTRUM

Product Note: Using the DDS Mode

Contents

Introducing the DDS Mode

Traditional AWG Mode

Before we take a deep dive into the various aspects of the newer DDS-mode, we first take a close look at the arbitrary waveform generator (AWG) mode in which Spectrum Instrumentation's signal generator devices traditionally operate. In this mode, the user pre-calculates the desired output waveform for each sample and sends it to the AWG memory. This data can then be played back to the analog output either once, in complex loops or in a continuous stream, depending on the chosen sub mode, like **Standard Replay-**, **FIFO-** or **Sequence-mode**. In principle, every type of output waveform is possible in this mode, making it a flexible solution for many use cases.

Storing long sequences of AWG data at high sampling frequencies requires a big onboard memory, that's why, for example, the M4i.66xx series of AWG cards with a maximum sampling frequency of 1.25 GHz has 4 Gigabyte (2 GSamples) of memory installed. This large memory enables it to store sequences longer than one second.

To transfer a continuous stream of data to the AWG, all Spectrum AWG cards have a high-speed PCIe interface, which can easily transfer Gigabytes of data per second. As continuously computing the required AWG data with a CPU can be quite challenging Spectrum Instrumentation cards also support Nvidia's state-of-the-art CUDA GPU accelerator cards using the SCAPP option.

In streaming FIFO-Mode, the onboard memory is used as a FIFO buffer for the incoming data. This can compensate for events where the data generating user application stalls for a moment, thus making the streaming very reliable. However, buffering also introduces latency, which can be disadvantageous in certain applications like closed feedback loops and general control systems.

For faster reaction time and reduced data transfer, the AWG has the **Sequence-mode**, where different prestored sequences can be looped in configurable order and switching between sequences can be triggered on an external trigger Input.

However, the Sequence-mode still uses a small FIFO buffer. Thus, if triggering a sequence change on external input, the old data in the FIFO buffer still needs to be transferred to the output and as the amount of old data in the FIFO buffer varies it can also introduce uncertainty in the dynamic behavior in this particular mode and use case.

DDS Mode

If only sinusoidal signals are needed the new **DDS-mode** can be used. Using Direct Digital Synthesis (DDS) continuous sinewaves can be generated in hardware on the AWG-card by only setting the desired **frequency** f_i, **amplitude** A_i and **initial phase** φ_i once. During runtime only changes need to be sent to achieve dynamic behavior.

Product Note: Using the DDS Mode

What is DDS?

Simplified, a **DDS core output $u_{Core,i}$(t)** performs the following calculation each **sample z** with the **sampling frequency f_s**:

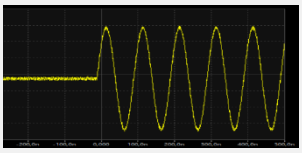
$$u_{Core,i}(t) = \sin(2\pi \cdot f_i \cdot t + \varphi_i) \cdot A_i$$

with

$$t = \frac{z}{f_s} \text{ and z = 0, 1, 2, 3 ...}$$

Example 1: if you just want to generate a continuous sine wave with 100 MHz and 90% amplitude of the DAC full-scale range only 3 specific commands need to be sent to the card using DDS mode:

```
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_AMP, 0.9);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_FREQ, 100.0e6);
spcm_dwSetParam_i32(hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);
```



Comparing DDS- and AWG-Mode

What advantages does DDS mode have?

DDS moves the task of calculating samples from your computing system to the FPGA on the generator-card. This can be very advantageous for closed feedback loops and continuous waveform data, that changes over time. As only changes need to be sent to the card, only small data packets need to be transferred and thus buffer lengths can be kept small to reduce response latency. As the data can be generated on-the-fly by the FPGA, it can react with deterministic latency to external trigger events while also keeping the phase of the signal continuous.

Furthermore, the math required to generate various sinusoidal signals can be quite challenging and error prone for many users. DDS reduces this complexity by offering an easy-to-understand high level command interface, which eases your programming.

What disadvantages does DDS mode have?

The DDS mode is intended for periodical sinusoidal signals, other signal forms can be approximated but generally the AWG is recommended in use cases needing waveforms other than sinusoids.

DDS has a fixed external Trigger-to-Output-timing with a jitter of 6.6 ns for unsynchronized trigger signals. In contrast, the standard replay modes have only a single sample jitter, e.g., 800 ps at 1.25 GS/s. So compared to the standard replay modes, the trigger-to-output jitter is 8 times larger at the full 1.25 GS/s sample rate.

DDS Mode in Detail

Multitone Signals

Instead of one basic DDS core the DDS module consists of up to N DDS cores[1] which outputs are all added together and output on one or more analog channels (Figure 1). Thus, so called multi-tone or multi-carrier signals can be produced which are vital for many applications in for example quantum research.

Simplified the **output u(t)** can be expressed as:

$$u_{CH0}(t) = \sum_{i}^{i=N} u_{Core,i}(t)$$

With i **= DDS core index**

The sum of all core amplitude values can add up to a maximum value of 1.0 or 100 %, if you set for example 2 cores both at 60 %, this will add up to 120 % and integer overflow effects occur at the output.

Example 2: To generate a range of 20 frequencies from 15 MHz to 90 MHz just program each core parameter one after another. To not exceed the output range of 100 %, an output amplitude of 5 % was chosen for all cores, except core 1, where 2.5 % instead was chosen to show the effect of different amplitude settings. The second carrier is thus a bit lower in amplitude than the others as can be seen in the spectrum of the signal (Figure 1).

```
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_AMP, 0.05);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_FREQ, 100.0e6);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE1_AMP, 0.025);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE1_FREQ, 120.0e6);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE2_AMP, 0.05);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE2_FREQ, 140.0e6);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE3_AMP, 0.05);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE3_FREQ, 160.0e6);
//... and so on
spcm_dwSetParam_i32(hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);
```

[1] Currently, the M4i.66xx series DDS-firmware has up to N = 20 cores on a single channel. Higher core number might become available in the future.

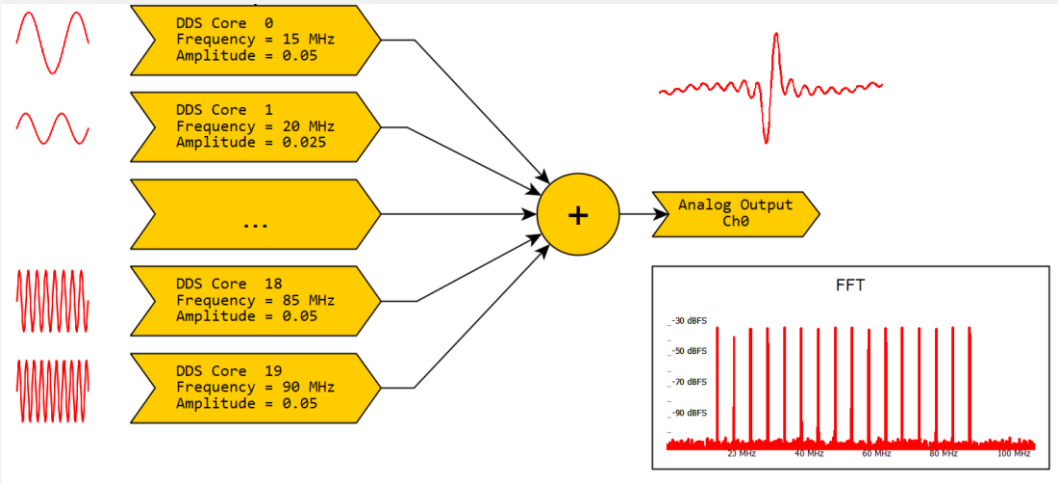


Figure 1: Summation of multiple DDS cores to one analog hardware channel.

Use of Pseudo Code in this product note

As Spectrum Instrumentation cards support different programming languages like C++ and Python this application note will use pseudo code to increase the readability and versatility of code snippets. Code Snippet 2 depicts the resulting pseudo code for the C++ Code Snippet 1.

```
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_AMP, 1.0);
spcm_dwSetParam_d64(hCard, SPC_DDS_CORE0_FREQ, 100.0e6);
spcm_dwSetParam_i32(hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);
```

Code Snippet 1: C++ code.

```
CORE0_AMP    = 1.0
CORE0_FREQ   = 100.0e6
CMD          = EXEC_AT_TRG
```

Code Snippet 2: Pseudo code.

Change of DDS Settings

All settings like frequency, phase and amplitude can be changed during runtime.

The changes can be queued sequentially, FIFO buffered, and executed simultaneously at a predefined trigger event using the EXEC_AT_TRIG command. Possible trigger events are:

- Card trigger, which supports the full trigger engine available to the card (see the user manual for setting up the trigger engine)
- Internal timer, which automatically triggers the next sequence at a predefined time interval.

Additionally, changes can be executed as soon as they reach the DDS module through the end of the queue using the EXECUTE_NOW command.

As the trigger source and timer interval can be changed just like all other parameters, creating complex sequences is an easy task.

The DDS module works on a fixed time base with a fixed timing resolution for all operations.

Example for calculating Trigger Resolution and Jitter:

The timing resolution for all trigger sources on M4i.66xx cards is t_{re} = 6.4 ns, which translates to 156.25 MHz.

The **internal timer,** for example, has a minimum value of 83.2 ns, but can be adjusted in 6.4 ns steps, so setting a timer value of (10 us) sets it precisely to 10.0032 us or 1563 x 6.4 ns. As all generator cards can be synchronized to an external clock source the timer can be just as accurate as your clock source.

If you use an **external trigger** to execute the next queued commands, the trigger is also detected with a timing resolution of 6.4 ns. If your trigger is asynchronous to the generator card, you will have a jitter of +/- 3.2 ns or 6.4 ns total. If your external trigger is synchronized to the generator card, and the phase was precisely set to always satisfy the sample and hold window of the trigger engine you can minimize the jitter to practically zero.

In general, it's worth mentioning that a group of generator cards can be easily synchronized using the Spectrum Instrumentation Starhub module with practically 0 ns jitter between devices.

Intrinsic frequency and amplitude slopes

In addition to frequency, amplitude and initial phase, the DDS module also supports linear frequency and amplitude slopes as dynamic parameters.

Example 4: Figure 2 shows an example of sequence programming with a linear slope.

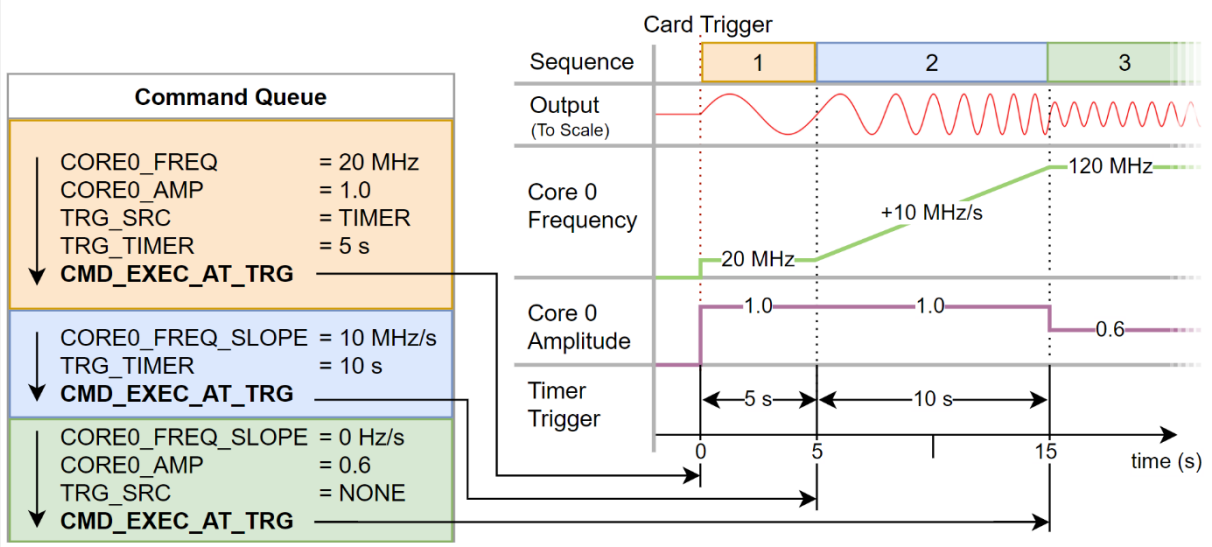


Figure 2: Frequency slopes and sequencing.

Sequence order:

0. Initially after reset, all parameters are set to 0 and the trigger source is set to the external card trigger.
1. At the card trigger event, the frequency is set to 20 MHz and amplitude to 1.0. To switch to the next sequence after 5 seconds, the trigger source is set to the internal timer, which starts counting to 5 seconds, where the next sequence is triggered.
2. The frequency of the core increases linear and smoothly by 10 MHz per second. Thus, after 10 seconds the final frequency of 120 MHz is reached.
3. The slope is stopped by setting it to 0 and simultaneously the amplitude is set to 0.6. Lastly, the trigger source is set to NONE, this disables the internal timer.

Quantization errors in Frequency and Amplitude slopes

While the slopes are practically linear, they are calculated inside the card's logic in discrete steps with quantized timing and value resolution. For the 66xx DDS module the minimum slope refresh rate t_{re} is 6.4 ns , which is more than sufficient for most applications for which they were designed.

Due to these quantized calculations, the slope increment and timing resolution is finite and a certain quantization error between the desired set value and the actual value might occur.

Typically, these effects only become visible with exceptionally long slopes in the order of seconds where the minimal slope step is reached or short slopes where the timing quantization becomes relevant. To mitigate these effects the final frequency should be set manually at the end of a slope.

Timing quantization error

The timing error is created, when entering a floating-point value for the timer, which is rounded by the driver to a full clock cycle in t_{re} steps. For about 1 millisecond long slopes this quantization error is in the range of +/- 3.2 ns or 3.2 parts per million and negligible for many applications. It can be mitigated by accounting for this quantization in your application and programming and only using time periods in t_{re} steps.

Behavioral timing Error

Furthermore, there is another timing error that can occur: With only a slope command at the execution event, the frequency is incremented simultaneously to the trigger event (Figure 3), leading to the expected end frequency.

However, when setting frequency and slope speed in the same execute event, the frequency is set in the first clock cycle and then the frequency is incremented in the next clock cycle as seen in Figure 4. While this behavior is technically not incorrect, the slope is therefore one clock cycle t_{re} shorter than might be expected. This behavior can be easily accounted for by setting the final frequency explicitly (Figure 5) or adjusting the start frequency by one increment.

Please note that a very short slope period of 4 clock cycles was chosen for display purposes, which are up to now not possible with EXEC_AT_TRIG commands as the minimal timer time is in the range of 100 ns. With slope periods in the order of milliseconds this effect is in the order of parts per million.

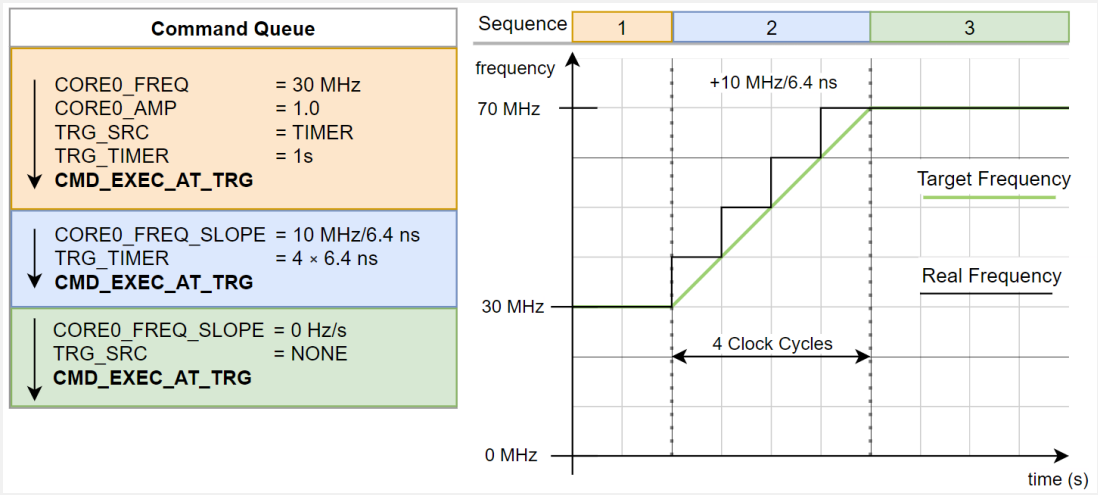


Figure 3: First the start frequency is set, secondly a slope is started.

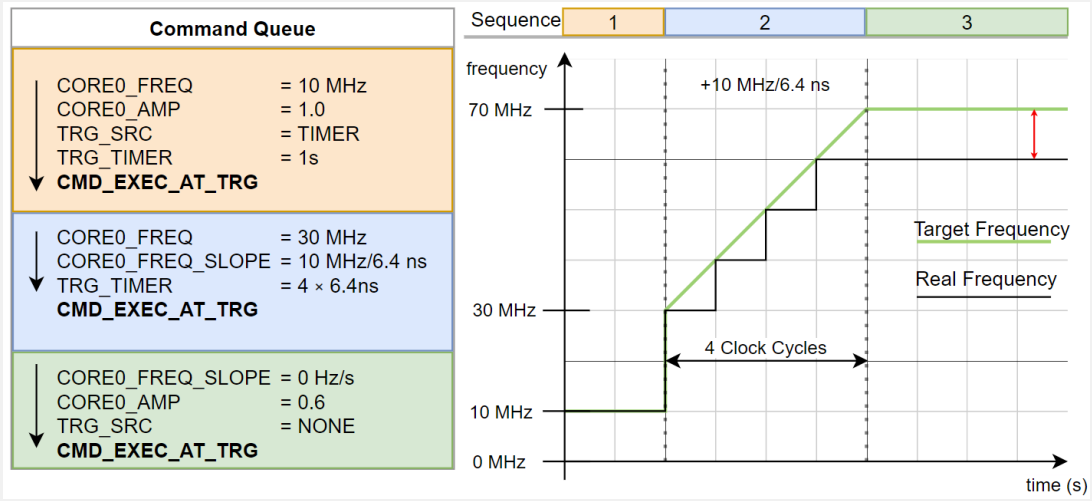


Figure 4: Frequency and Slope set in same execution event resulting in different end value.

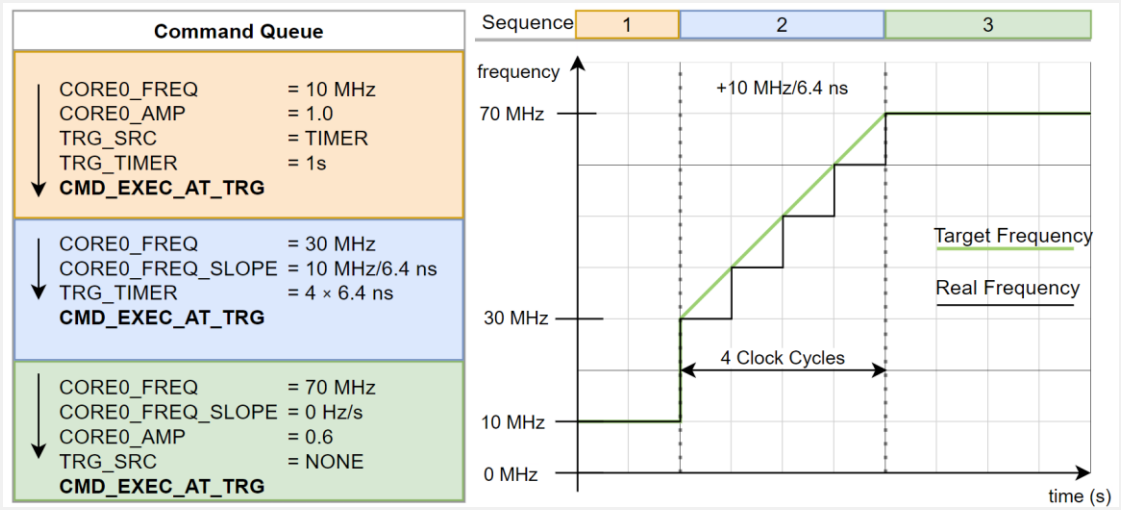


Figure 5: Frequency and slope set in same execution event; end value is set explicitly.

Value quantization Error

To reduce the value error slopes are calculated internally with a higher slope $\dot{f}_{step}$ resolution than the frequency resolution f_{step} of the DDS to reduce the quantization error. For longer time periods in the order of seconds, a step divider can be set which increases the resolution further but is typically not necessary for applications in the timescale of milliseconds.

$\dot{f}_{step}$ can be read in the API by reading the SPC_DDS_AVAIL_FREQ_SLOPE_STEP Register and f_{step} by reading the SPC_DDS_AVAIL_FREQ_STEP register.

Example:

The 66xx DDS module has a frequency resolution f_{step} of round about 0.3 Hz . If you would set the slope speed exactly to f_{step}/t_{re} the frequency would increase every 6.4 ns by exactly 0.3 Hz. If this

would also be the minimum frequency slope resolution $\dot{f}_{step}$, a slope value lower than 0.3 Hz / 6.4 ns wouldn't be possible and lead to no slope performed at all.

After one second this minimal value would lead to a frequency change of 47 MHz. So, after one second only quantized frequency changes of 0 Hz, 47 MHz, 2 * 47 MHz, 3 * 47 MHz and so on would be possible.

Summarized; to calculate the maximum frequency slope quantization error you just multiply the slope resolution times the slope duration. We divide by two because we round to a lower or higher value in between values:

$$f_{error} = \frac{\dot{f}_{step} \cdot \Delta t}{2}$$

The actually implemented frequency slope resolution $\dot{f}_{step}$ is higher than f_{step} with $\dot{f}_{step}$ = 694 Hz/s. So, if you want a slope from 1 MHz to 2 MHz in one millisecond you will end up with a maximum error of $\frac{\dot{f}_{step} \cdot \Delta t}{2} = 0.348\ Hz$, which is negligible for most application it was designed for.

The following code snippet performs a slope from 100 MHz to 101 MHz in 1 millisecond and ends up at 101.00000014 MHz.

```
// Set initial Values, keep them for one second
TRG_SRC           = TIMER
TRG_TIMER         = 1 s
CORE0_AMP         = 1.0
CORE0_FREQ        = 100 MHz
CMD               = EXEC_AT_TRG

// Start the Slope
TRG_TIMER         = 1 ms
CORE0_FREQ_SLOPE  = 1 MHz / 1 ms
CMD               = EXEC_AT_TRG

// Stop the Slope
CORE0_FREQ_SLOPE  = 0
TRG_SRC           = NONE
CMD               = EXEC_AT_TRG
```

Phase behavior

As can be seen in Figure 2 the phase remains continuous while changing the frequency. This can be explained by the more accurate mathematical representation of the DDS core including a **phase accumulator $\theta_i(z)$**:

$$u_{Core,i}(z) = \sin(\theta_i(z) + \varphi_i) \cdot A_i$$

$$\theta_i(z) = \theta_i(z-1) + 2\pi \cdot \frac{f_i}{f_s}$$

On reset or phase jump:

$$\theta_i(0) = 0$$

As can be derived from the formula, changing the frequency f_i only changes the phase increment $2\pi \cdot \frac{f_i}{f_s}$ per clock cycle, while the phase accumulator output only changes indirectly with time. Setting the frequency to 0 for example halts the phase, while in a phase coherent DDS the phase would jump.

Phase Shift Mode

Per default, changing the initial phase φ_i from 51° to 52° will change the phase of the output signal by 1° relative to the previous state. This mode behaves like a typical benchtop device, where you can adjust the phase continuously by rotating a knob. Possible applications would include modulation schemes like Phase-Shift-Keying (PSK).

Phase Coherent Operation / Phase Jump Mode

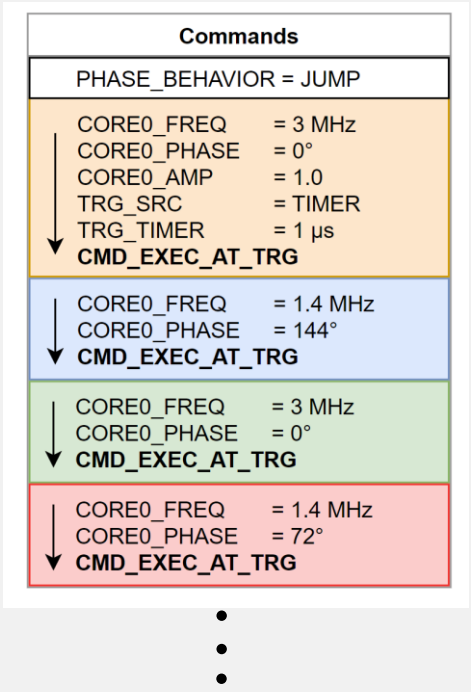
Some applications involve phase coherent frequency switching, where the phase jumps each time the frequency switches. In this time-multiplex manner, multiple frequencies can be output simultaneously using a single DDS core. Practically, this has the advantage of requiring a lower dynamic range from the analog path compared to summing all frequencies simultaneously.

As described earlier, the phase is normally continuous, but the DDS module also features a phase jump mode for coherent operation, which resets the phase accumulator $\theta_i(z) = 0$ each time the phase is set.

Example 5:

Figure 6 shows 2 different frequencies which are time division multiplexed to one single DDS core. Averaged over time the frequency spectrum will show both frequencies, although only 1 core is being used.

In real world applications, it's advisable to choose a switching frequency much lower than the required signal frequencies, to reduce the effect of switching spurs.



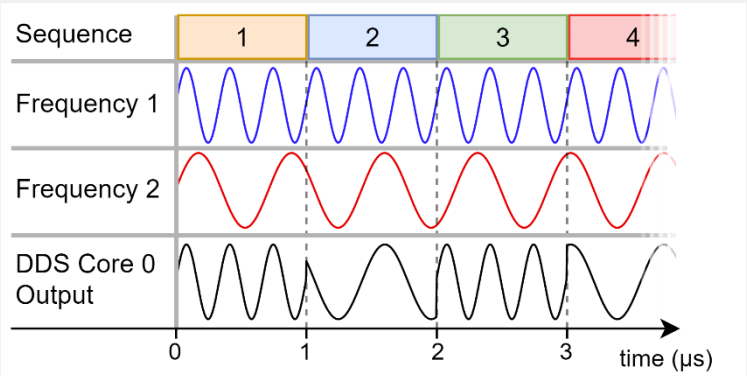


Figure 6: 2 different frequencies generated on a single output.

To achieve this behavior both frequency and phase need to be set at the switching point. To calculate the required phase in ° at the switching points only a simple calculation is needed:

$$\varphi(t) = f \cdot t \cdot 360° \; mod \; 360°$$

$$\varphi(1s) = 1.4\,MHz \cdot 1\,\mu s \cdot 360° \; mod \; 360° = 144°$$

$$\varphi(2s) = 3\,MHz \cdot 2\,\mu s \cdot 360° \; mod \; 360° = 0°$$

$$\varphi(3s) = 1.4\,MHz \cdot 3\,\mu s \cdot 360° \; mod \; 360° = 72°$$

Everything in between is calculated in hardware. Thus, switching between frequencies can easily be done with switching frequencies in the range of 100 kHz for all DDS cores, effectively multiplying the number of tones that can be generated by the DDS.

Custom frequency slopes

Although the DDS module only supports linear slopes, other slope types can be easily approximated by using a finite number n of linear frequency slopes which form a piecewise linear interpolation of the desired shape. For example, many applications in the cold atom physics world require sinusoidal, so called "s-shaped", frequency slopes (

Figure 7). Depending on the smoothness required by the use case the user can choose a specific number of segments. For display purposes about 7 segments seem to match the target as depicted in

Figure 7. In the following, we'll explain how these slopes can be calculated and programmed.

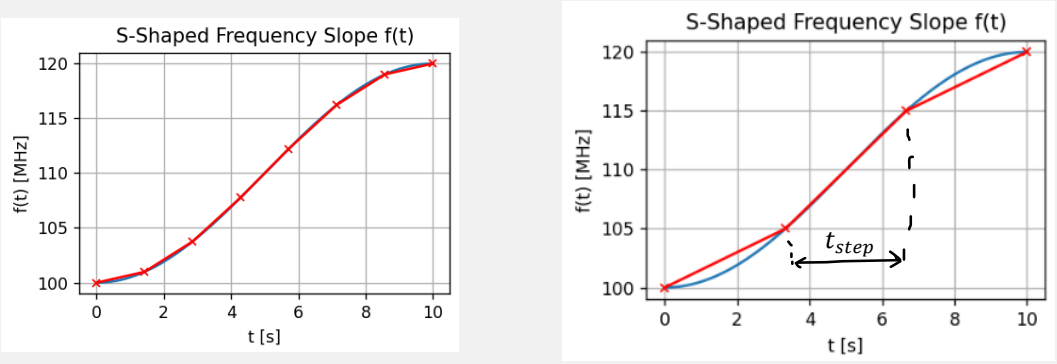
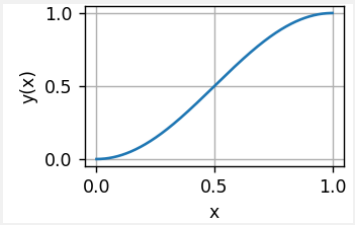


Figure 7: Approximated waveform with n = 7 segments and actual waveform behind the approximation

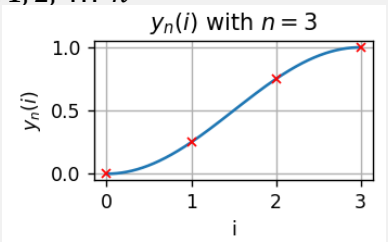
1. First, we need to define our desired s-shape function by using a sinusoidal function, with x = 0 being the beginning of the slope and x = 1 the end of the slope:

$$y(x) = \frac{-\cos(x \cdot \pi) + 1}{2}$$



2. Next, we adjust its range to the number of steps we want and quantize it to full integer steps. This is done by substituting $x = \frac{i}{n}$ with $i: 0, 1, 2, \ldots\ n$

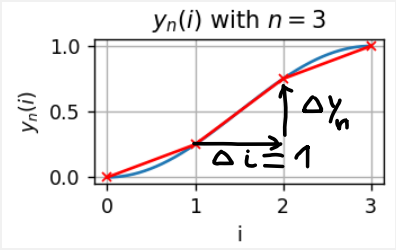
$$y_n(i) = y\left(\frac{i}{n}\right)$$



Product Note: Using the DDS Mode

3. Calculate the unitless gradient $M(i)$ of each linear line segment with i = 0, 1, 2, … n-1:

$$M(i) = \frac{\Delta y}{\Delta i} = \frac{y_n(i+1) - y_n(i)}{1}$$



These values could be precalculated and stored in a Look Up Table for faster execution.

4. With $M(i)$ we can build a simple function $s(i, f_{delta}, t)$, which will scale $M(i)$ to the specific frequency difference Δf and time difference Δt:

$$s(i, \Delta f, \Delta t) = M(i) \cdot \frac{\Delta f}{\Delta t}$$

$$\Delta f = f_{end} - f_{start} = 120\ \text{MHz} - 100\ \text{MHz} = 20\ \text{MHz}$$

5. Define the initial frequency and hold it for 5 seconds at the beginning:

```
CORE0_FREQ      = f_start
TRG_SRC         = TIMER
TRG_TIMER       = 5s
CMD             = EXEC_AT_TRG
```

6. Set the time t_{step} between the piecewise linear Δi sequences:

```
TRG_TIMER       = t_step
```

7. Set the slope for each step with i = 0, 1, 2, … n-1:

```
for (i = 0; i < n; i++)
  {
  CORE0_FREQ_SLOPE   = f_start + s(i, Δf, Δt)
  CMD                = EXEC_AT_TRG
  }
```

8. Lastly, we set the final frequency, stop the slope, and stop the internal timer:

```
CORE0_FREQ          = f_end
CORE0_FREQ_SLOPE    = 0
TRG_SRC             = NONE
CMD                 = EXEC_AT_TRG
```

Controlling XIO-Lines using DDS Commands

DDS supports 3 different XIO Output Modes:

- **Manual**:
  In this mode you can control the output state, high or low, just like the frequency of a DDS core.
- **Waiting for trigger**:
  The XIO line serves as a status signal indicating whether the DDS module waits for a trigger, it goes high after an Execute-At-Trig is received and the DDS module starts waiting for a trigger. It goes low after it receives the programmed trigger signal (timer or card trigger).
- **Execute**:
  The XIO line serves as a status signal, it goes high each time a trigger or execute now command is received and new parameters are loaded to the output.

All signals are aligned to the output, see Figure 15 for an example.

So, if you want to change a XIO line set in “Manual”-mode from low to high and change the output amplitude from 0 to 100 % at the same trigger event, both changes can be measured roughly at the same time. Simultaneously to the change of the amplitude and the manual XIO lines, you will see the XIO-line set to “Execute”-mode going high for 6.6 ns and then going low. The “Waiting for trigger”-line will simultaneously go low as the trigger arrives and go high as soon the next “Execute-At-Trig” command is send to the card.

Software driven feedback loops / decision making.

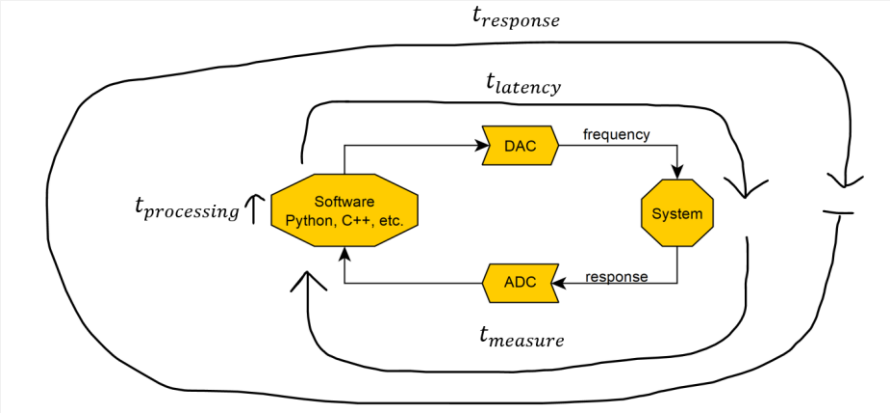


Figure 8: Example Feedback Loop and timing.

Imagine a closed feedback loop comprised of a generator card (AWG), a black box system and a digitizer. Within this loop, the AWG stimulates the system using frequency f, and the system responds by either signaling our system to quickly increase the frequency or to lower the frequency.

In this example, the new frequency value must reach the system within a certain response time $t_{response}$. This response time is depicted in Figure 8 and comprises three different main components:

$$t_{response} = t_{measure} + t_{processing} + t_{latency}$$

Where:

- $t_{measure}$ is the time it takes to measure the changed input signal.
- $t_{processing}$ is the time to calculate the new required output parameters/data.
- $t_{latency}$ is the latency from the software setting the new parameters to the card output.

If we know the required frequency for a trigger event beforehand, we can preload the frequency value for the next trigger event and react easily within a microsecond. This is the "Trigger-to-Output" time. However, in this real time feedback application, we don't know beforehand which output frequency is needed next.

In other words, we need to make a decision how to react after we receive the new input parameter and perform the reaction in time.

Previously, with the AWG FIFO mode this meant that new data had to be calculated first and then pushed into AWG memory. Due to buffering, this introduces a lot of latency in the system (typically on the order of several milliseconds). With AWG sequence mode, this time could be reduced by switching prestored sequences. The switching process can, however, introduce time uncertainties (on the order of 100s of nanoseconds).

With DDS mode the processing time can be decreased, as only the newly required output frequency needs to be calculated and not a long sequence of individual samples. Furthermore, as the data to transfer is minimal, the latency time can also be drastically reduced as only minimal buffering is needed.

To recap chapter 0 the new DDS mode supports 2 different execution commands: EXEC_NOW, where the new frequency value can be loaded to the output as fast as possible and EXEC_AT_TRIG where the new value is loaded precisely on a selected trigger source with well-defined timing. So, to reach the fastest response time, EXEC_NOW can be used, whereas to have deterministic timing and in particular latency, EXEC_AT_TRIG needs to be used. We will examine this further in the following section.

Measuring the response time with EXEC_NOW

Measuring the latency time $t_{latency}$ from software to physical output is quite difficult. Imagine measuring the time from pressing a key to the character being displayed. The start time defined by your finger press is quite imprecise when measuring in the order of milliseconds.

However, we can easily measure the full response time $t_{response}$, so how long it takes for the processing system to detect a changed input by polling it in software and then sending a new frequency value to output another signal. Now instead of an ADC we use two auxiliary digital IO lines, which can already be found on M4i cards. If IO line XIO-0 goes high, we increase the frequency as fast as possible and if XIO-1 goes high, we decrease the frequency. Please note that the XIO lines are not used as trigger inputs and are just polled by software.

This parameter was tested for different systems and programming languages. The results for a few single commands can be seen in Table 1: Latency measuring results. However, as with all non-real time operation systems these latency values are not guaranteed. It's always possible that, for example, a virus-scanner stalls the system, and the latency increases dramatically! That's why It's best to test this value with your own system using the "Detailed Test Setup" described in the next section.

Table 1: Latency measuring results.

Condition	Typical achievable Roundtrip Times
C++ on Windows 10, Intel i7 13700k	20 µs
Python on Windows 10, Intel i7 13700k	70 µs

Detailed Test Setup:

As a signal source for the IO lines XIO-0 and XIO-1 we use two push buttons. To clearly see the result of the change introduced by the reacting software, we use another IO line XIO-2 as an output, which we toggle from low (0) to high (1), and an analog output, which signal frequency we change.

Finally, the time delay between the IO line XIO-0 or XIO-1 going high, and the output signal being changed is what we're interested in. We can display and calculate this time by using an oscilloscope.

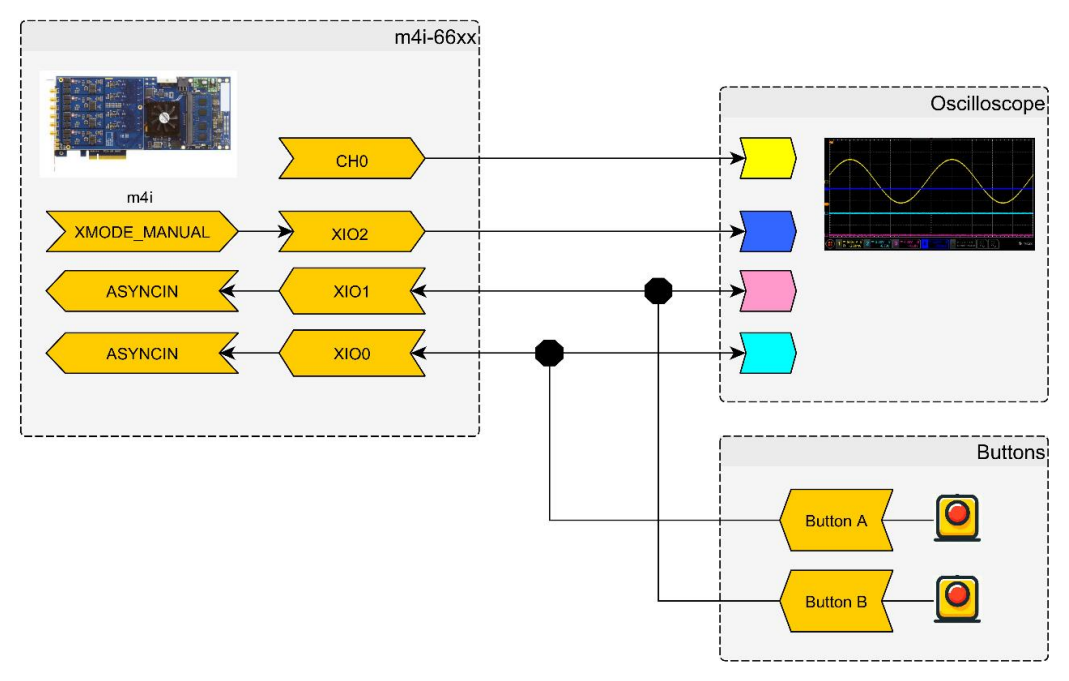


Figure 9: Test Setup for Software to Output Latency Test

```
// Send initial Values
X0_MODE = ASYNC_IN
X1_MODE = ASYNC_IN
X2_MODE = DDS_MANUAL

CORE0_AMP      = 1.0
CORE0_FREQ     = 0.5 MHz
MANUAL_OUTPUT  = X2 -> 0
CMD            = EXEC_AT_TRG
CMD            = WRITE_TO_CARD
M2CMD_CARD     = FORCETRIGGER

// Poll XIO Lines until either one is High
while (1)
  {
  if (get_param(XIO0) == 1)
        {
        CORE0_FREQ      = 0.1 MHz
        CORE0_AMP       = 1.0
        MANUAL_OUTPUT   = X2 -> 1
        CMD             = EXEC_NOW
        CMD             = WRITE_TO_CARD
        break;
        }
  else if (get_param(XIO1) == 1)
        {
        CORE0_FREQ      = 1 MHz
        CORE0_AMP       = 1.0
        MANUAL_OUTPUT   = X2 -> 1
        CMD             = EXEC_NOW
        CMD             = WRITE_TO_CARD
        break;
        }
  }
```

Code Snippet 3: Pseudocode for the Software to Output Latency Test

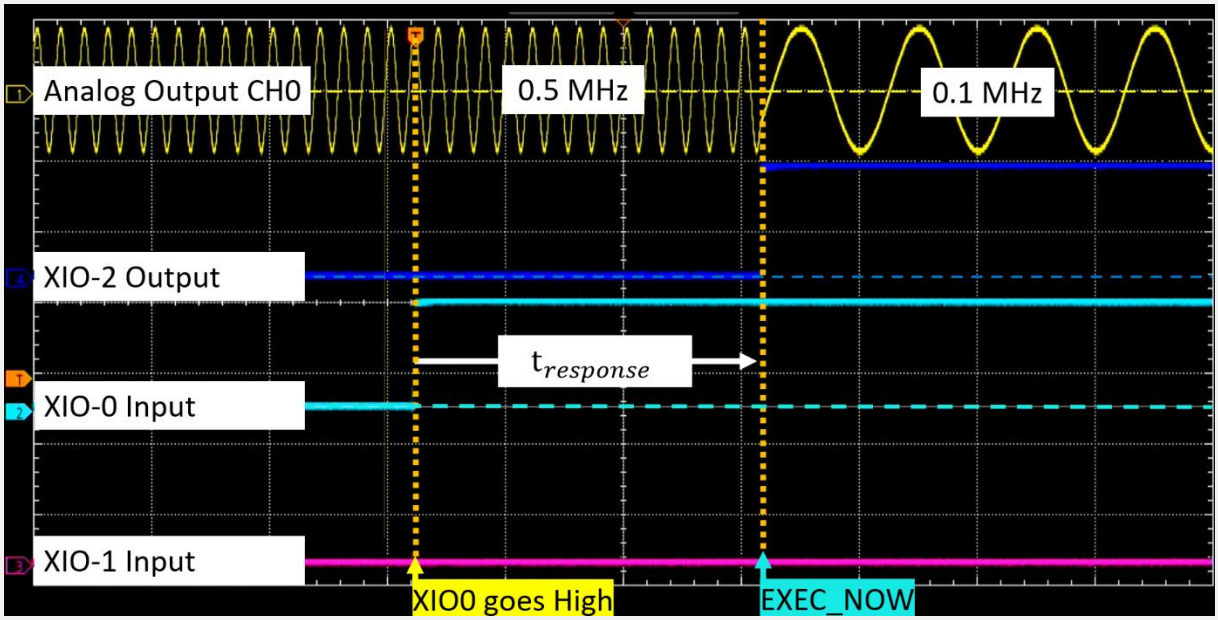


Figure 10: Scope Picture resulting from a Software to output latency test. Button A was pressed.

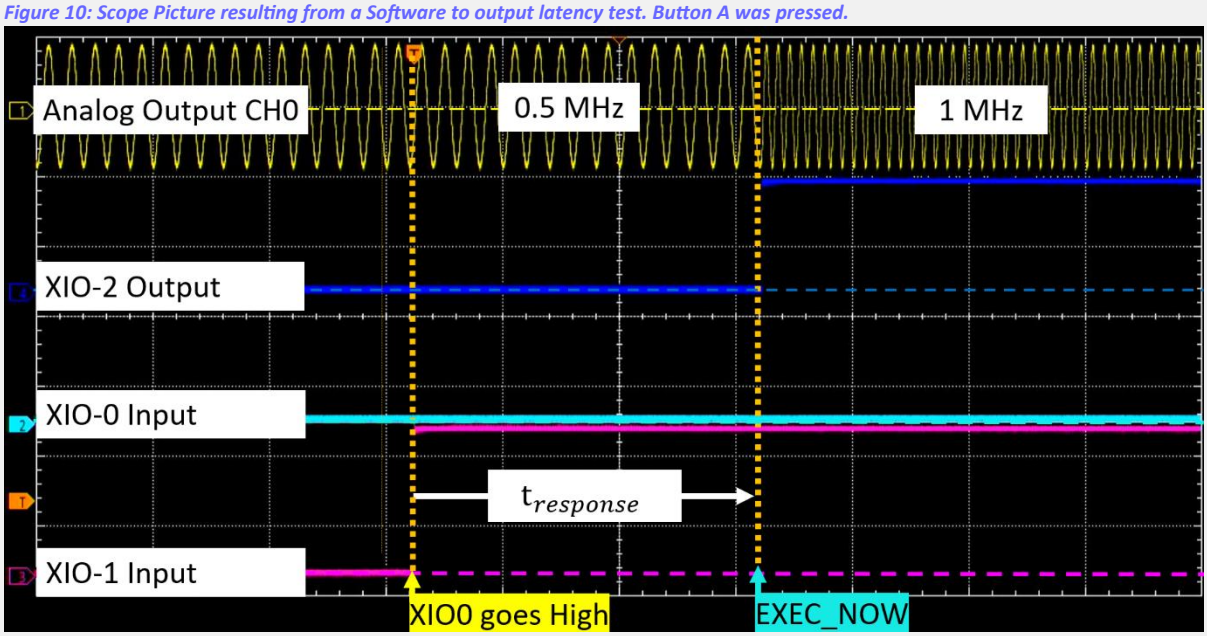


Figure 11: Scope Picture resulting from a Software to output latency test. Button B was pressed.

Achieving deterministic latency with EXEC_AT_TRIG

The latency from an external trigger to the dynamic change of the output frequency can be fixed, by using two trigger events with fixed delay between them. The first trigger event is used to trigger the readout of the new input parameters and the second trigger to execute the new frequency value.

For example, if you know your software response time to an external trigger event varies between 20 µs and 80 µs max, you can set the second trigger event 100 µs after the first one. The new

frequency values will now always be output 100 µs after the first trigger event. For the second trigger event you can use an external trigger again or the built-in timer.

Test Setup

We now need to use a different approach to select the next frequency, as the M4i.66xx series of cards doesn't support XIO lines as trigger inputs. So, we use one push button connected to the trigger input TRG0 of the generator card and connect a switch to the IO line XIO0. If the switch is in position high on a trigger event generated by the push button, we want frequency A output and if its low frequency B. So, we first need to set the switch to the desired XIO-0 level and then push the button. If you control these signals digitally you can of course toggle the XIO-0 line simultaneously to the trigger.

Regarding the programming, instead of polling if an XIO line has changed, we use the card's trigger engine. We set it to trigger if TRG0 goes high. The DDS in turn has a status signal "WAITING_FOR_TRIG", which goes low if the trigger was received. We poll this signal to go low and then check whether XIO-0 reads back 0 or 1. Depending on this we set the value for the next trigger event.

To achieve the deterministic latency with the second trigger event, select and setup the built-in timer trigger to 100 µs with the first trigger event.

The IO-line value is only evaluated once somewhere between the timepoints where the trigger is pressed, and the timer trigger executes the next sequence. The X-marker in Figure 13 and Figure 14 thus mark a signal region of "Don't Care".

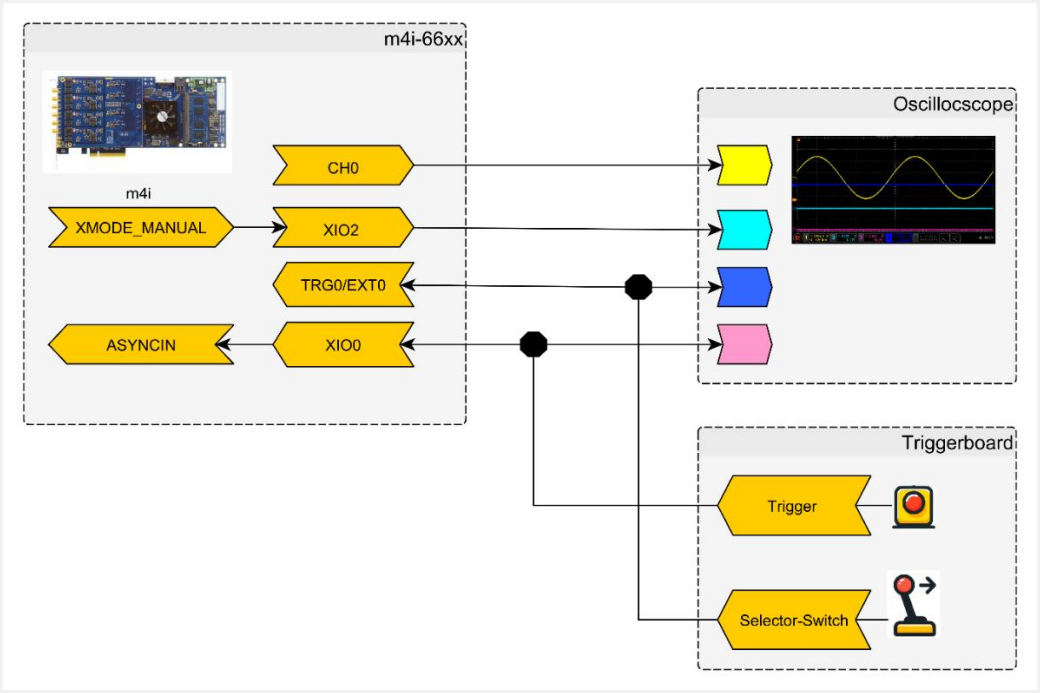


Figure 12: Test setup for deterministic latency with decision making.

Product Note: Using the DDS Mode

```
frequency_initial        = 100 kHz
frequency_a              = 20 kHz
frequency_b              = 300 kHz

X0_MODE        = ASYNC_IN
X2_MODE        = DDS_MANUAL
// Setup the card trigger engine to trigger on EXT0
TRIG_ORMASK    = SPC_TMASK_EXT0

// Send initial Values
CORE0_AMP      = 1.0
CORE0_FREQ     = frequency_initial
MANUAL_OUTPUT = X2 -> 0

CMD            = EXEC_AT_TRG
CMD            = WRITE_TO_CARD
// Force Trigger initial settings
M2CMD_CARD     = FORCETRIGGER

/* This yellow marked EXEC_AT_TRG is triggered by the external
Trigger and starts the timer for the NEXT Blue EXEC_AT_TRG */
DDS_TRIG_SRC   = TIMER
TIMER           = 100µs
CMD            = EXEC_AT_TRG
CMD            = WRITE_TO_CARD

// Wait until we received the trigger
while (get_param(WAITING_FOR_TRIG) == 1)
    {
    // Wait
    }

/* Send new Data based on IO Line state.
The Data needs to be send before the timer runs out */
if (get_param(XIO0) == 0)
    {
    CORE0_FREQ       = frequency_a
    MANUAL_OUTPUT    = X2 -> 1
    CMD              = EXEC_AT_TRG
    CMD              = WRITE_TO_CARD
  }
else
  {
    CORE0_FREQ       = frequency_b
    MANUAL_OUTPUT    = X2 -> 1
    CMD              = EXEC_AT_TRG
    CMD              = WRITE_TO_CARD
    }
```

Code Snippet 4: Pseudo code for decision making with deterministic latency.

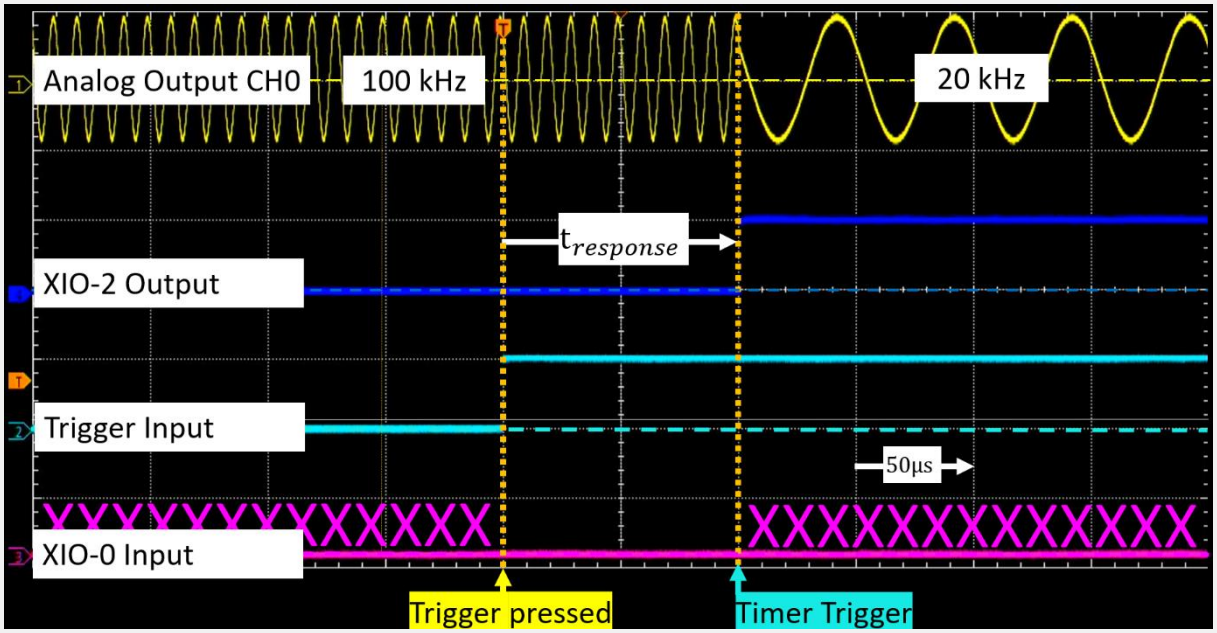


Figure 13: Scope picture for deterministic latency with decision making test. Decision A, XIO-0 is low on external trigger event. $t_{response} = 100\mu s$

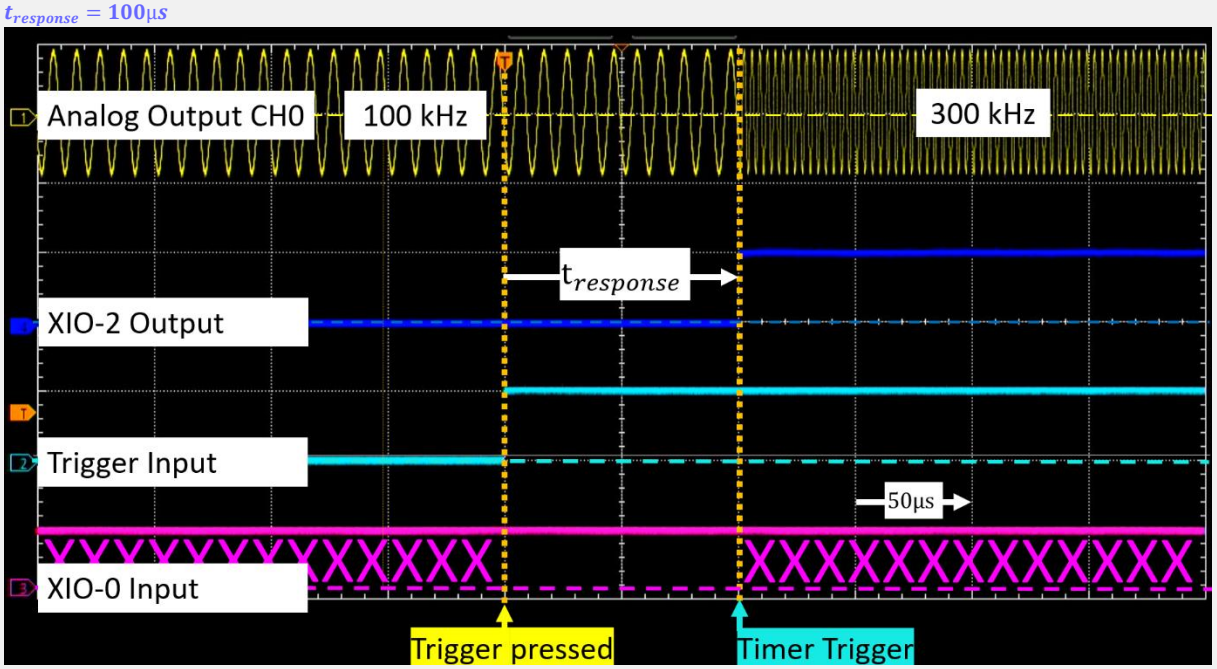


Figure 14: Scope picture for deterministic latency with decision making test. Decision B, XIO-0 is high on external trigger event. $t_{response} = 100\ \mu s$

Continuous Streaming

Let's imagine you want to create your own FM radio station at 101.3 MHz and use the AWG card as a Software Defined Radio. As mentioned beforehand, previously you've had to stream data for each sample resulting in a data stream of over 2 GB/s at 1.25 GHz sample rate! This usually mandated a dedicated CUDA GPU accelerator card.

Now with DDS only the frequency modulation signal needs to be sent, consisting of new frequency values. For audio signal with 20 kHz bandwidth a sampling rate of 44.1 kHz is usual. This signal can be streamed continuously to the DDS requiring only minimal computing power.

To increase the reliably of the stream a large FIFO buffer can be used. The Spectrum API fits each command into just 64 bits, or 8 bytes. This allows up to 500 million commands to be stored in the card's 4 GB on-board memory! This will however also increase the latency of the stream. With the buffer full at 44.1 kHz command rate this will result in a latency of just over 3 hours! But you can always choose the maximum number of commands you want to buffer.

Maximum Streaming Speed:
The streaming speed is highly dependent on the user application and system. Tests with a Python implementation on the M4i.66xx reached reliable speeds of about 300 kHz, while in C++ continuous streaming command rates of over 10 MHz were tested successfully.

Single Step Programming using Execute-Now

Imagine you want to generate a continuous signal of 100 MHz for a lengthy period of 20 seconds and then add a short arbitrary signal in between that changes every few nano seconds. Unfortunately, the timer has a minimal re-arm time of around 100 ns. However, with a more complex "single instruction" per clock cycle approach you can change the output signal every 6.4 ns.

This is possible by using single Execute-Now instead of Execute-At-Trig Commands. These commands are executed immediately when they arrive at the end of the command queue. Thus, they have different timing behavior depending on the state of the queue:

- If they are sent to the card with the queue empty and the card not waiting for any Trigger, they are executed as fast as possible defined by the "Software to Output latency".
- If two Execute-Now-commands are queued after another, they are executed 6.4 ns apart.
- If an Execute-Now-command is queued after an Execute-At-Trig-command, they are executed with a fixed delay time t_1 after the trigger.

Example 7:

Let us suppose we want to output a sine wave after a trigger signal, but also want to control a digital device using XIO output 2. The device expects a specific sequence of 0’ and 1’ as depicted in the Figure 1. Below is the according command queue.

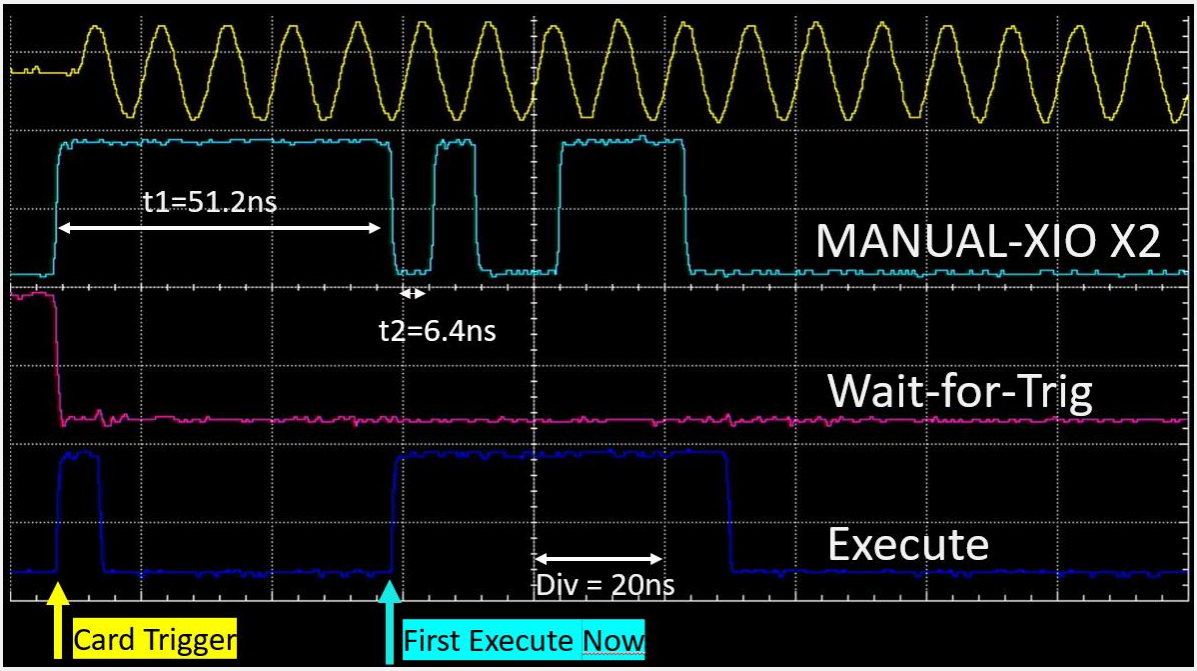


Figure 15: Different XIO output modes and programming using single “Execute-Now” commands

Command List

```
CORE0_AMP          = 0.7
CORE0_FREQ         = 30 MHz
MANUAL_OUTPUT   = X2 -> 1
CMD             = EXEC_AT_TRG

MANUAL_OUTPUT   = X2 -> 0
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 1
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 0
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 0
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 1
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 1
CMD             = EXEC_NOW
MANUAL_OUTPUT   = X2 -> 1
CMD             = EXEC_NOW
MANUAL_OUTPUT    = X2 -> 0
```

Conclusion

Comparing DDS And AWG on the M4i-66xx Card

The AWG-mode is perfect if your output signal can be precalculated and stored into memory. If not, you need to loop it, stream it, or use sequence mode, making the setup a bit more complex. If you have a closed loop feedback system you need to stream the data with high latency or use the sequence mode.

If your signal consists of sine waves, you just need to describe it using the DDS Commands. You're not limited by the internal card buffer and don't need to calculate buffer sizes. Latency in the command FIFO can be kept quite short as the amount of data that needs to be buffered can be extremely low. Switching between DDS settings during runtime (sequences) has a fixed Ext. Trigger to Output Latency time. Continuous data generation doesn't require much computational power, as only changes need to be sent.

	AWG-Modes	**DDS**
Signal Types	Arbitrary or repeating	Sinusoidal
Starhub Compatible	Yes	Yes
Output Data Generation	In Software by User	In Hardware
Ext. Trigger to Output Latency[1]	397.2 ns	554 ns
Ext. Trigger to Output Latency Jitter	- 1 Sample Clock (800 ps) for the start of the output - In Sequence Mode: about. 600 ns min. for switching between prestored sequences.	6.4 ns (8 Sample Clocks) for all externally triggered events
Sampling Frequency	Adjustable but limited by memory Bandwidth. 1.25 GHz max. on 2 channels 625 MHz max. on 3 and 4 channels	1.25 GHz fixed on all channels.

[1]At 1.25 GHz sampling rate

Summary

The DDS firmware option supports a broad range of applications due to its high flexibility. It can reduce latency, thus enabling more use cases than the traditional AWG mode. Furthermore, it eases programming, simplifying existing use cases, where the AWG mode was previously used.

All-in-all, the M4i.66xx series Spectrum Instrumentation's cards support countless use cases for signal generation cards.

Author

- Arnold Walker, M. Sc. – Development Engineer at Spectrum Instrumentation GmbH

