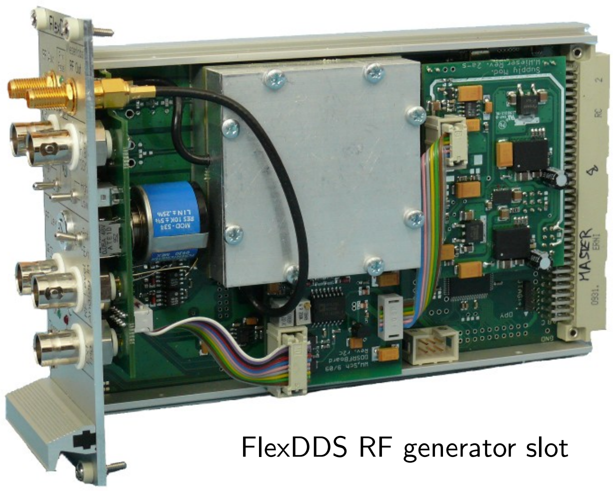
FlexDDS

(Rev. 2c_2, Nov 2011)

Flexible Multi-Channel Phase-Coherent RF Source

Features

- Multi-channel operation with precisely known and adjustable phase relationship between channels
- Real-time control of all signal parameters
- Phase-continuous frequency tuning
- Computer interface (e.g. USB 2.0 link, RS-232)
- Optional per-slot processor (see PDCPU below)


FlexDDS RF generator slot

Components

- **FlexDDS-Rack:** 19" rack which integrates the computer interface and power supplies. The rack can hold up to 8 independent **FlexDDS** RF generator slots and 1 frontpanel controller slot (**FlexDDS-FPCtl**).
- **FlexDDS:** RF generator slot module
- **FlexDDS-FPCtl:** Slot module for reference clock and trigger.

Full rack

General Description

FlexDDS is a multi-channel phase-coherent RF source. The design deliberately targets the needs of experimental physicists who want to control all signal parameters in real-time from a computer. Initially, a series of actions (like changes in amplitude or frequency, start of frequency sweeps,. . .) is compiled into commands which are then transferred to the FlexDDS-Rack via a USB link. Each time a (real-time asynchronious) trigger input is activated, FlexDDS-Rack executes one or several commands and waits for the next trigger event. There is no limit on the number of successive commands as they are loaded continuously from the host computer.

One outstanding feature of FlexDDS is its defined and known phase relationship between channels. For example, two channels can easily be set up to produce an RF output at the same frequency and with equal phase. Slightly detuning the frequency of one channel will then linearly increase the phase difference between the two channels.

Basic Operation

The DDS rack reads commands sent via the USB/RS232. The binary format for both interfaces is the same and the document will refer only to USB. The commands are sent as a stream of 2-byte (16 bit) values, transferred as LSB first (!). See the next chapter for a description of the data transfer to the DDS. See also the example near the end of the document.

After reset, the DDS starts reading data from the USB. If there is no data available, it waits for the host computer to supply data (yellow LED on). Since revision 2c_2, FlexDDS has an internal FIFO buffer for up to 8192 words of data (either USB or RS232). The buffer (FIFO) is completely transparent to the user.

The host computer will now typically configure the slots (write frequency tuning words, ramp registers etc.) Once the host has configured the slots, it can set the continuation bit to zero while specifying which slots should receive the next trigger. It is important to understand that the DDS registers (such as the frequency tuning word) are only stored in the AD9910 DDS chip but are not yet active. To activate the new values, a trigger signal is needed.

As soon as a 2-byte word with continuation bit 0 is read, the rack waits for a trigger to arrive. (Yellow LED off.) The trigger is routed to selected slots activating the previously written values. The trigger is typically the BNC trigger input of the rack (tiny leftmost control slot: top BNC input) and triggers on the rising edge (falling edge can be configured). Alternatively, the red key on the leftmost control slot can be pressed to manually generate a trigger. The green LED on the control slot flashes quickly once the trigger is delivered. At the same time, the red trigger LEDs on the individual slots which get triggered with this particular trigger pulse also flash quickly.

Directly after the trigger, the DDS rack resumes reading data from the USB and/or the internal FIFO buffer.

Individual slots can also be triggered via the red trigger pushbutton on each individual slot. As alternative to the trigger pushbutton, the slot can be configured to accept an individual trigger via one of the BNC inputs. The use of individual triggers is discouraged unless necessary. By default, indivudially triggering one slot as outlined in the last sentences does not make the rack continue reading from the USB port. This can be changed by configuring the backplane trigger bus.

A powerful feature on a per-slot basis is the profile (Gray) counter: The 8 single tone profiles (each consisting of frequency, phase and amplitude) can be loaded into the DDS slot. Using BNC inputs on the frontpanel slot, different profiles can be activated on a counter-like basis: 2 BNC inputs can be used: One to count the profile conter up/down and one to select the counting direction (optional). This way, frequency or phase can be changed very quickly – much faster than the 500kHz update rate of the slot writes.

The ramp generator built into the AD9910 can be used to perform frequency, phase and amplitude ramps in up/down direction and optionally turn back once reaching the limit. The FlexDDS slots can be configured so that BNC inputs on the slot can be used to hold the ramp or change the direction of the ramp in real time.

More powerful actions can be carried out via the RAM modulation feature of the AD9910. For a full documentation, please consult the AD9910 datasheet from Analog Devices.

Synchronization

The DDS has a 10MHz external clock input. This input can be used as reference clock for all DDS channels. Best signal is a 1-2V square wave although smaller sine waves work as well.

The trigger is internally synchronized to the 250MHz DDS sync clock. A synchronized copy of the trigger signal is routed back via the SMA output on the rack control slot (the tiny leftmost slot) with programmable polarity and 3.3V logic level. The first edge of this trigger output is synchronized to the DDS.

Note that there is a latency between the trigger and the update. It is recommended to switch on the "matched latency" bit in the CFR2 register of the AD9910 (set by default in FlexDDS) so that changes in amplitude, phase and frequency appear on the output at the same time.

Jitter-free triggering can be achieved by ensuring that the trigger signal is coherent with the 10MHz reference clock *and* ensuring that the rising trigger edge has an appropriate delay with respect to the next rising reference clock edge. This delay can be adjusted by inserting/removing a couple of centimeters cable in one of the cables (trigger or refclock).

For synchronization of other devices, a sync trigger output is provided by FlexDDS (SMA output on the tiny control slot). For each trigger, a short pulse is delivered at this output. The first edge of this pulse is synchronized to the internal DDS sync clock (250MHz).

Data Transfer to the DDS

Data is sent as a 16bit wide stream via the USB FIFO interface. The FPGA respects USB handshake, so it waits for new data to arrive from the host if the data stream stalls. (The RS-232 interface, if used, must be configured for RTS/CTS hardware handshaking for the same reason.)
The 16 bits have the following meaning:

```
15      8 7     0
 C......L DDDDDDD
```

C – (bit 15) continuation bit: data is read as long as this is 1. If 0, read is stopped until a trigger is received.
L – (bit 8) local bit:
1: data is processed by the FPGA in the rack
0: data is sent to the DDS slots
D – (bits 7..0) 8 data bits

Depending on the "local" bit (8), the bits denoted with '.' have different meaning.

Local case: L=1 (bit 8 is 1)

```
15      8 7      0
 C....AA1 DDDDDDDD
```

A – (bit 10..9) local register address
00: (`CMD`) execute a command
01: (`CS_DATA`) select which slots receive the following data
10: (`CS_TRIG`) select which slots receive the following trigger signals
11: (`ICFG_BUS`) internal FPGA/rack configuration bus

`CMD` **register:** Writing to this register executes a command:

00000001:	Send a (synthetic) trigger pulse.
00010010:	Store the value in the `ICFG_BUS` register into the `ICFG0` register (most important rack control register).
00100010:	Store the value in the `ICFG_BUS` register into the `ICFG1` register (do not change this register; internal use).
00000011:	Make rack read from USB. DO NOT USE. (For internal use only.)

`CS_DATA`, `CS_TRIG`**:** Slot selection bitmask registers.

If a bit is set (1), the corresponding slot receives the next non-local write (`CS_DATA` register) or the next trigger event (`CS_TRIG` register).

bit 0:	leftmost (master) slot (next to the control slot)
...	
bit 7:	rightmost slave slot

The `ICFG` registers are the most important rack config registers. Normally, there is no need to change the bits in there. In order to change the ICFGx register, first execute a write to the `ICFG_BUS` register (see above), then execute the "write ICFGx register" command (e.g. for `ICFG0`, write "00010010" into `CMD` register).
`ICFG0` register bits:

bit 0	unused
bit 1	unused
bit 2	Force the use of the local 10 MHz clock even if an external 10 MHz reference clock is supplied to the rack,
bit 3	Enable trigger from control slot (BNC and key) (Normally 1)
bit 4	Enable backplane trigger source (from DDS slots). (Normally 0)
bit 5	Enable LVDS IO with frontpanel slot (normally 0, using normal single-ended IO).
bit 6	Invert rack trigger input. (Normally 0, i.e. rising edge triggered.)
bit 7	Invert the synchronized trigger output on the frontpanel (SMA out). If set, falling edge, otherwise rising edge (default).

Note: ICFG0 has reasonable default after reset (clocks enabled); do not change it unless you know what you are doing.

`ICFG1` register bits:

bit 0	internal use
bit 1	If set, lock out USB from access to FIFO; RS232 not affected.
bit 2	Disable the differential master refclock from the rack to the DDS slots.
bit 3	Deactivate 10 MHz crystal in rack.
bit 4	If set to 0, reset the USB FIFO chip.
bit 5	If set, clear the FIFO buffer in the rack.
bit 6	internal use
bit 7	unused

Note: ICFG1 has reasonable default after reset and is not meant to be changed by the host computer.

Remote case: L=0 (bit 8 is 0)

```
15       8 7       0
 C....RR0 DDDDDDDD
```

R – (bit 10..9) remote destination bits
- 00: (`DDS`) write DDS registers (AD9910) (dds_sel=0, trig=0)
- 01: (`FPGA_DATA`) write slot FPGA register content (dds_sel=1, trig=0)
- 10: currently unused / reserved
- 11: (`FPGA_ADR`) set slot FPGA register address (dds_sel=1, trig=1)

In order to write DDS registers, set `RR=00` and stream the DDS data in the DATA section (D) using several writes to transfer DDS register address followed by DDS register content. Example:

```
1....000 00000000      <- select CFR1 register in the AD9910 (address 0)
1....000 00000000      <- first byte (MSB) to write into the CFR1
1....000 01000000      <- second byte (bit 22 enables the inverse sinc filter)
1....000 00000000      <- third byte
1....000 00000000      <- fourth byte (LSB completing the 32bit register write)

1....000 00001001      <- select ASF register in the AD9910 (address 9)
1....000 00011000      <- first byte (MSB) to write into the ASF
1....000 00111000      <- second byte  (ampl. ramp rate set to 6200)
1....000 11111111      <- third byte   (ampl. scale factor to be set to 16383)
1....000 11111100      <- fourth byte (LSB completing the 32bit register write)
```

In this example, dots '.' denote unused bits which should be written as `0`.

These register write commands get routed to those slots which were selected by the currently active content of CS_DATA (0x83 0x..).

For DDS register addresses and register content in the AD9910 DDS chip please refer to the AD9910 datasheet available at Analog Devices (www.analog.com). Note: Do not change comminucation-related parameters in the DDS (like MSB vs. LSB first format or the SDIO pin direction). Do not change the AD9910 PLL and/or multi-channel synchronization settings unless you know precisely what you are doing. These registers are set up by the internal controller after a reset.

Upon startup, the rack controller sets up reasonable defaults. The PLL settings and the multi-chip synchronization should not be changed unless necessary. The single tone profile 0 (STP0) is set up with 0 MHz sine, phase offset 0, full amplitude (effectively produces no output because the frequency is 0).

The slot FPGA itself has a number of configuration registers as well. To write these registers, use RR=11 and RR=01: First, set the register address you wish to write to using RR=11, then perform a register write using RR=01. See further down for a slot FPGA register description.

FlexDDS slot FPGA internal register map

This section documents registers in the FPGA of each individual FlexDDS slot. These registers may not be confused with the registers found in the AD9910 DDS chip.

OSK_CTLA, OSK_CTLB:	control the OSK pin of the AD9910 DDS
DRCTL_CTLA, DRCTL_CTLB:	control the DRCTL pin of the AD9910 DDS
DRHOLD_CTLA, DRHOLD_CTLB:	control the DRHOLD pin of the AD9910 DDS
BPTRIG_CTLA]	control the SLOT-to-backplane trigger
PROFILE_CTLA,PROFILE_CTLB:	control the PROFILE0,1,2 pins of the AD9910 DDS

Register Name	Address	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
reserved	0x00								
BPTRIG_CTLA	0x01	ffset	ffclr	omux0	fmod0	iinv	isel2	isel1	isel0
reserved	0x02								
reserved	0x03								
OSK_CTLA	0x04	omux1	omux0	fmod1	fmod0	iinv	isel2	isel1	isel0
OSK_CTLB	0x05	0	0	0	0	0	0	ffset	ffclr
reserved	0x06								
reserved	0x07								
DRCTL_CTLA	0x08	omux1	omux0	fmod1	fmod0	iinv	isel2	isel1	isel0
DRCTL_CTLB	0x09	0	0	0	0	0	0	ffset	ffclr
reserved	0x0a								
reserved	0x0b								
DRHOLD_CTLA	0x0c	omux1	omux0	fmod1	fmod0	iinv	isel2	isel1	isel0
DRHOLD_CTLB	0x0d	0	0	0	0	0	0	ffset	ffclr
reserved	0x0e								
reserved	0x0f								
PROFILE_CTLA	0x10	cinv	csel2	csel1	csel0	dinv	dsel2	dsel1	dsel0
PROFILE_CTLB	0x11	0	0	0	cval2	cval1	cval0	cload	cclr
reserved	0x12								
reserved	0x13								

```
SLOT_SCFG0    0x14      0     0     0     0     0      0    oksmpe bpsck
TRIG_CTLA     0x15      0     0     0     0     0    ifppt  enfpt  entgk
-------------------------------------------------------------------------------
```

Bit description of `OSK_CTLA, DRCTL_CTLA, DRHOLD_CTLA`

The OSK pin controls the DDS output: If LOW, the RF output is off, if HIGH the RF output is switched on.
The DRCTL pin controls the direction (up/down) of a DDS ramp (e.g. frequency ramp) while DRHOLD can be used to hold the ramp generator if HIGH.
The FPGA engine for these 3 registers follows the same pattern: Two control registers `CTLA` and `CTLB` control how their corresponding pin is used. The following list refers to the `CTLA` register.

bit(7..6): omux(1..0):
Control the output multiplexer attached directly to the corresponding DDS pin (OSK, DRCTL, DRHOLD).

11:	force output HIGH
10:	force output LOW
01:	edge triggered output
00:	transparent level output

bit(5..4): fmod(1..0)
Operation mode of the flip-flop; only effective if omux(1..0) is set to edge triggered output.

11	(reserved)
10	toggle with each input transition
01	set output HIGH at first intput transition (rising edge triggered)
00	set output LOW at first intput transition (rising edge triggered)

bit(3): iinv
Input invesion bit; if set the input (specified by isel(2..0) is inverted before being passed to the output or to the flip-flop. By inverting the input, the flip-flop is made falling edge triggered instead of rising edge triggered.

bit(2..0): isel(2..0)
Input selector.

111	frontpanel comparator input ("Comp. In (10MHz)")	[FP_IN_CMP]
110	frontpanel TTL A input ("TTL In A")	[FP_IN_C]
101	frontpanel TTL B / OSK input ("TTL In B (OSK)")	[FP_IN_B]
100	frontpanel trigger input ("Ext. Trig")	[FP_IN_A]
011	DDS RAM_SWP_OVR pin used as input	
010	DDS DROVER pin used as input	
001	update trigger input	
000	default value (LOW for DRHOLD, HIGH for OSK and DRCTL)	

(Names in the right column are internal signal names and meaningless for the end user.)

Bit description of `OSK_CTLB, DRCTL_CTLB, DRHOLD_CTLB`

bit(7..2):
reserved; write them as 0

bit(1): ffset

bit(2): ffclr
If written 1, the flip-flop is set (ffset) or cleared (ffclr), respectively. This can be used to prepare the internal flip-flop state for edge-triggered operation.
If both ffset and ffclr are set, clear takes precedence.

Examples

For example, if you wish to have the "TTL In B (OSK)" input on the slot frontpanel as OSK pin (to digitally switch on/off the RF output), use the following register contents:

```
OSK_CTLA = 0000i101    (i = 0 or 1, deciding whether LOW or HIGH is "ON")
```

This is actually the default setting after power-up.
To use the "TTL In A" input on the slot frontpanel as DDS sweep ramp direction pin, use:

```
DRCTL_CTLA = 0000i110  (i = 0 or 1, deciding whether LOW or HIGH is "UP")
```

To force the DRCTL pin LOW and use the "TTL In A" input on the slot frontpanel as DDS sweep ramp hold pin, use:

```
DRCTL_CTLA  = 10000000
DRHOLD_CTLA = 0000i110  (i = 0 or 1, deciding whether LOW or HIGH is "HOLD")
```

You can also use the rising edge of the "TTL In A" to stop a ramp, so that the ramp will not resume even after the falling edge. (HIGH state on DRHOLD holds the DDS ramp.)

```
DRHOLD_CTLA = 0101i110   (rising edge will bring flip-flop into HIGH state)
DRHOLD_CTLB = 00000001   (clear the flip-flop to LOW state)
```

The inversion ('i' bit) may become necessary to establish the desired polarity (depending on whether an inverting opto coupler or comparator is used in the internal signal path of the FlexDDS).
NOTE: For these examples to work, the AD9910 has to be configured correctly, e.g. OSK has to be enabled.

Bit description of `BPTRIG_CTLA`

This register can be used to control the backplane trigger signal. The backplane trigger signal is a wired-OR signal on the backplane which allows the slots to generate triggers. Care has to be taken that if two slots generate a trigger at the same time, only one trigger will be "seen" by the backplane control unit. Also, make sure that these triggers do not coincide with frontpanel triggers.

bit(7): ffset

bit(6): ffclr
If written 1, the flip-flop is set (ffset) or cleared (ffclr), respectively. This can be used to prepare the internal flip-flop state for edge-triggered operation. If both ffset and ffclr are set, clear takes precedence.

bit(5): omux(0):
Control the output multiplexer attached directly to the trigger pulse generator.

1:	edge triggered output
0:	transparent level output

bit(4): fmod(0)

Operation mode of the flip-flop; only effective if omux(0) is set to edge triggered output.

1:	toggle with each input transition
0:	set output HIGH at first intput transition (rising edge triggered)

bit(3): iinv

Input invesion bit; if set the input (specified by isel(2..0) is inverted before being passed to the output or to the flip-flop. By inverting the input, the flip-flop is made falling edge triggered instead of rising edge triggered.

bit(2..0): isel(2..0)

Input selector. See `OSK_CTLA`.

Bit description of `PROFILE_CTLA`

The registers PROFILE_CTLA and PROFILE_CTLB control the FPGA output to the AD9910's PROFILE(2,1,0) pins. These pins allow to quickly change the profile (i.e. frequency+phase+amplitude) between 8 different settings. The handling in the FPGA consists of a loadable up/down Gray counter. By default, profile 0 is active. Different sources can be selected both for the counter as well as for the up/down selection.

Note: The counter is a Gray code counter with the following counting sequence: 0,1,3,2,6,7,5,4 ("upward" counting)

The counter wraps around and re-starts after 8 steps.

bit(7): cinv

bit(6..4): csel(2..0)

Select counter clock input into the 3 bit counter which controls the profile pins. The cinv bit can be used to invert the input (count on falling edge rather than rising edge). The csel(2..0) selects the source; the bit values are identical with those used in OSK_CTLA:isel(2..0).

bit(3): dinv

bit(2..0): dsel(2..0)

Select up/down counting action. By default, the counter counts upwards (HIGH), but you can use sources to change the counting direction. dsel(2..0) selects the up/down source; the bit values are identical with those used in OSK_CTLA:isel(2..0). The default value (isel=000) is HIGH. dinv inverts the source, so setting isel=000 and dinv=1 will select downwards counting instead of default upwards.

Bit description of `PROFILE_CTLB`

bit(7..5):

Reserved; write them as 0

bit(4..2): cval(2..0)

These bits represent the counter value of the 3 bit counter which controls the PROFILE pins. If cload is set, the value is stored in the counter.

bit(1): cload

Write this to 1 if you wish to store the value cval(2..0) in the 3 bit counter. (See errata!)

bit(0): cclr

If set, the counter is cleared. If both cclr and cload are set, the outcome is undefined. (See errata!)

Examples

To count profiles up like 3 -> 2 -> 6 -> 7 -> ... (Gray sequence) triggered by the positive edge of the "TTL In A" frontpanel input:

```
PROFILE_CTLB = 00001110   (load counter with start value 3 (See errata!))
PROFILE_CTLA = 01100000   (configure rising edge UP counting with ‘‘TTL A’’)
```

To be able to change the counting direction using the "TTL In B" frontpanel input and to count using the falling edge of the "TTL In A" frontpanel input:

```
PROFILE_CTLA = 11100101
```

("TTL In B": HIGH/LOW for UP/DOWN counting)

Bit description of `SLOT_SCFG0`

This is an internal config register for the slot and should not be changed by the host computer. A useful default is set upon reset.

bit(7..2):
: Reserved; write them as 0

bit(1): oksmpe
: When set, select SYNC_SMP_ERR (multi channel sync status) for the green "OK" LED on the slots; when cleared select PLL_LOCK (PLL lock indicator) for the green "OK" LED.

bit(0): bpsck
: Disable sync clock to the backplane control board. Usually switched on (0) for the master slot and off (1) for all other slots.

Bit description of `TRIG_CTLA`

Trigger configuration register.

bit(7..4):
: Reserved; write them as 0

bit(3): enbpt
: Enable the backplane trigger. This trigger signal is used by the backplane control unit to update the DDS after writing DDS registers. If this is disabled, the registers will still be written, but no update pulse will be delivered to the DDS from the backplane. In order to make the new register values appear on the DDS output (e.g. to make a frequency change effective), either a local trigger pulse (via the frontpanel key or input - if enabled) or a PROFILE change has to be performed as a substitute for the backplane trigger pulse. Enabled (1) by default. Only disable if you know what you are doing.

bit(2): ifppt
: Invert frontpanel trigger input. This can be used to change from rising edge triggered to falling edge triggered. Ineffective unless bit(1) is set. Disabled (0) by default.

bit(1): enfpt
: Enable triggering of the slot via the frontpanel trigger input. Disabled (0) by default.

bit(0): entgk
: Enable triggering of the slot via the frontpanel trigger key (1). Enabled (1) by default.

Parallel Data CPU (PDCPU, PDCPUv2) Firmware Add-On

Note: This section only applies to those FlexDDS slots which have the PDCPU firmware add-on installed. There are two version of this add-on: The original version and the second version (v2). The PDCPUv2 has all features of the original PDCPU plus two extras: Extended delay and access to the serial data bus into the AD9910 DDS register to change DDS registers. Those parts which only apply to PDCPUv2 are marked with "v2 only" in the documentation.

The PDCPU add-on is installed in the FPGA of the FlexDDS slots. It installs a RAM of 8192 words and a processor ("CPU") to execute instructions stored in the RAM. Each instruction is consumes word of the RAM and is executed at a rate of 1 instruction per clock. The clock rate is currently 31.25 MHz (i.e. 1 GHz/32). The RAM word size is 18 bits.

```
Register Name Address  Bit7   Bit6   Bit5   Bit4   Bit3   Bit2   Bit1   Bit0
------------------------------------------------------------------------------
PDRAM_ADRL    0x20     <--            RAM address low byte                -->
PDRAM_ADRH    0x21     <--            RAM address high byte               -->
PDRAM_DATA    0x22     <--            RAM data (sequential write)         -->
PDRAM_CTLA    0x24       0      0    crst   txdis  txen   cpudis cpuen  rrst
PDRAM_CTLB    0x25       0      0     0      0      0     reserv reserv reserv
------------------------------------------------------------------------------
```

Writes to these registers are carried out like regular slot FPGA register writes (0x86 address, followed by: 0x82 data; see below).

CPU instructions are written to the RAM in the following way:

1a Set RAM address LOW byte by writing to `PDRAM_ADRL`

1b Set RAM address HIGH byte by writing to `PDRAM_ADRH`
The valid RAM addresses are in range 0..8191 and select the address of the instruction (18 bit word) to be written.

2a Write the instruction LOW byte (bits 0. . . 7) to `PDRAM_DATA`

2b Write the instruction MID byte (bits 8. . . 15) to `PDRAM_DATA`

2c Write the instruction HIGH bits (bits 16. . . 17) to `PDRAM_DATA`
The last write (2c) actually commits the instruction to the RAM. The RAM address pointer set in 1a,1b is now automatically incremented by 1 so that the next word in RAM can be written by resuming at step 2a.

For example, to go to RAM address 0x0105 and write to instructions, the following data has to be sent over the USB link.

```
    1....110 00100000    <-- Select slot FPGA register address 0x20 (PDRAM_ADRL)
    1....010 00000001    <-- Write PDRAM_ADRL register (address HIGH byte, 0x01)
    1....110 00100001    <-- Select slot FPGA register address 0x21 (PDRAM_ADRH)
    1....010 00000101    <-- Write PDRAM_ADRL register (address LOW byte, 0x05)
    1....110 00100010    <-- Select slot FPGA register address 0x22 (PDRAM_DATA)
    1....010 aaaaaaaa    <-- Write instruction LOW byte (aaaaaaaa)
    1....010 bbbbbbbb    <-- Write instruction MED byte (bbbbbbbb)
    1....010 000000cc    <-- Write instruction HIGH bits (cc)
```

```
             The 18 bit instruction "cc bbbbbbbb aaaaaaa" is now stored in RAM
             at address 0x0105. Address pointer incremented to 0x0106.
    1....010 eeeeeeee    <-- Write instruction LOW byte (eeeeeeee)
    1....010 ffffffff    <-- Write instruction MED byte (ffffffff)
    1....010 000000gg    <-- Write instruction HIGH bits (gg)
             The 18 bit instruction "gg ffffffff eeeeeee" is now stored in RAM
             at address 0x0106. Address pointer incremented to 0x0107.
```

Bit description of `PDRAM_CTLA`

Control register A. This is the main parallel data CPU control register and allows to enable/disable/reset the CPU.
Note: Only those bits which are written as 1 will carry out any action. Zero bits do not change anything. E.g. writing only cpudis will only disable the CPU and not affect the `TXENABLE` pin.

bit(7..6):
Reserved; write them as 0

bit(5): crst
CPU reset. Writing this bit to 1 will reset the CPU. The bit automatically flips back to 0 after less than 1 μs. Resetting the CPU will set the CPU's execution address pointer to 0, abort any wait and clear the wait register, zero the port A register (port D is not affected).

bit(4): txdis
Writing this bit to 1 will disable the `TXENABLE` pin to the AD9910. May not be set together with bit `txen`.

bit(3): txen
Writing this bit to 1 will enable the `TXENABLE` pin to the AD9910. May not be set together with bit `txdis`.

bit(2): cpudis
Disable the CPU if this bit is written as 1. The CPU should be disabled before modifying the RAM content or resetting the CPU. May not be set together with bit `cpuen`.

bit(1): cpuen
Enable the CPU if this bit is written as 1. The CPU should be enabled after the RAM has been written. As soon as the CPU is enabled, it starts executing instructions from the RAM. May not be set together with bit `cpudis`.

bit(0): rrst
When written as 1, reset the RAM filling all RAM contents with zero.

Bit description of `PDRAM_CTLB`

Control register B. Internal use only; do not access this register.

CPU Instructions

The CPU understands the following instructions:

```
Instructions to write to the 18bit parallel data bus into the AD9910 DDS:
  00 AAAAAAAA AAAAAA00   Set 14 bit DDS amplitude value A  (AD9910: F1,0=00)
  01 PPPPPPPP PPPPPPPP   Set 16 bit DDS phase value P      (AD9910: F1,0=01)
  10 FFFFFFFF FFFFFFFF   Set 16 bit DDS frequency offset F (AD9910: F1,0=10)
  11 AAAAAAAA PPPPPPPP   Set 8 bit DDS amplitude A and phase P     (F1,0=11)

Instructions interpreted by the PD CPU:
  00 0aaaaaaa aaaaaa11   JUMP to address aaaa.
  00 ........ ..000001   NOP (no operation, consumes 1 cycle delay)
  00 rrrrrrrr mm000101   WAIT (m: wait mode, r; wait register)
  00 ........ s.001001   UPDATE (s: now or next)
  00 vvvvvvvv pp001101   WRITE to PORT p with value v
  00 cccccccc nn010001   CONDition c (nn is reserved and must be 00)
  00 dddddddd L.010101   SPI: Transfer data d over SPI into AD9910 (v2 only)
  00 ....wwwe ..011001   WAIT_XREG: wait extension registers (v2 only)
```

Above dots ('.') denote "don't care bits". All other illegal instructions are ignored like the NOP instruction.

Program termination: If the program counter reaches the end of the RAM it wraps over to address 0. This means, if the program should stop at some point (e.g. end of the program), it is recommended to use a WAIT with a condition that never comes true or a JUMP to the same address.
Hint: You can simply store the instruction "JUMP 8191" at RAM address 8191 which is the end of the RAM. This prevents the program counter from wrapping over to address 0 and execute the program again. At the end of the program, simply place a "JUMP 8191" as well. If you do not do so, the CPU will execute all instructions between program end and RAM end which will usually be all zero (i.e. instruction to set the DDS amplitude to 0).

Parallel Data Write Instructions

The first 4 instructions listed in the instruction summary above are data write instructions. When executing one of these instructions, the CPU simply writes the passed data to the AD9910's 16+2 bit parallel data bus. Basically, the first two bits are the `F1` and `F0` pins of the AD9910 while the remaining 16 bits are the 16 parallel data pins.

Since the DDS amplitude has 14 bit precision only, the DDS amplitude *must* be written with the two least significant bits zero! All other instructions understood by the CPU have the same form as the DDS amplitude write instruction except that the last two bits are *not* zero. This allows space-efficient instruction storage.

CPU Instruction `JUMP`

This instruction makes the CPU unconditionally jump to a new RAM address. The next instruction executed by the CPU will be the one found at the new RAM address. The RAM address in range 0...8191 is encoded in bits 2 to 14 named 'a'.

CPU Instruction `NOP`

This instruction does nothing, simply consumes one clock cycle. This can be used to deliberately insert delays.

CPU Instruction `WAIT`

This instruction can wait for a specified amount of time or for some event to happen.
The `WAIT` instruction has 3 different forms:

```
00 rrrrrrrr 00000101   WAIT DELAY SHORT
00 rrrrrrrr 01000101   WAIT DELAY LONG
00 Irrrrrrr 10000101   WAIT FOR EVENT
00 ........ 11000101   WAIT SPI (wait for SPI writes to complete; v2 only)
```

`WAIT FOR EVENT` waits until an event happens. Here, the 7 'r' bits make up a bitmask. Each bit can select one independent event source:

bit 6	frontpanel comparator input ("Comp. In (10MHz)")	[FP_IN_CMP]
bit 5	frontpanel TTL A input ("TTL In A")	[FP_IN_C]
bit 4	frontpanel TTL B / OSK input ("TTL In B (OSK)")	[FP_IN_B]
bit 3	frontpanel trigger input ("Ext. Trig")	[FP_IN_A]
bit 2	DDS RAM_SWP_OVR pin used as input	
bit 1	DDS DROVER pin used as input	
bit 0	update trigger input	

If any one of the selected bits is HIGH, then the wait condition becomes true, otherwise it is false.
Depending on the bit 8 ('I'), the CPU stops waiting and resumes execution when the condition is true (I=0) or when it is false (I=1).

`WAIT SPI` (v2 only) waits for the SPI FIFO buffer to be entirely written into the AD9910 via the serial SPI port. See the `SPI` PDCPUv2 instruction below.

`WAIT DELAY SHORT` waits for the number of cycles secified with the 'r' bits (0. . . 255) plus an additional delay of 3 cycles.

PDCPUv2: The additional delay is 4 cycles. If the `WAIT_XREG` instruction is used before the `WAIT`, the delay is further extended. See below table for a summary of delays.

`WAIT DELAY LONG` waits for the number of cycles secified with the 'r' bits *multiplied with 256*, plus the same additional delay similar to `WAIT DELAY SHORT`.

PDCPUv2: The additional delay is 4 cycles. If the `WAIT_XREG` instruction is used before the `WAIT`, the delay is further extended. See below table for a summary of delays.

The following table summarizes wait timings for different `WAIT` instructions. In the table, N specifies the value of the delay byte (`dddddddd`, range 0. . . 255) and w specifies the predivider value for the `WAIT_XREG` instruction (v2 only; range 0. . . 7) when used at all. One cycle is 32 ns.

Instruction(s)		Wait time in cycles (32 ns)	PDCPU version
`NOP`		1	all
	`WAIT SHORT`(N)	$N + 4$	all
	`WAIT LONG`(N)	$256 \times N + 4$	all
`WAIT_XREG`(w)	`WAIT SHORT`(N)	$2^{(2w+2)} \times (N + 2.5) + 2$	v2 only
`WAIT_XREG`(w)	`WAIT LONG`(N)	$2^{(2w+2)} \times (256 \times N + 2.5) + 2$	v2 only

NOTE: The table above does not count the time consumed by executing the `WAIT_XREG` instruction itself (add 1 cycle for that).
Any delay below 4 cycles can be reslized by 1 to 3 `NOP` instructions.
Long delays with high precision can be realized by using several delay instructions, e.g. first a coarse `WAIT LONG` with `WAIT_XREG` followed by a fine-grained `WAIT SHORT`.

CPU Instruction `UPDATE`

This instruction sends an UPDATE to the AD9910 DDS chip. If the bit 7 ('s') is set, then the instruction will send the update one CPU cycle later (`UPDATE_NEXT`). This is especially useful to set the signal phase when "auto-clear phase" is set in the AD9910 (`CFR1` bit 13).

CPU Instruction WRITE

This instruction writes to one of the ports of the CPU and has the following form:

```
00 vvvvvvvv 00001101   WRITE to PORT A
00 vvvvvvvv 11001101   WRITE to PORT D
```

The 8 bit value specified as 'v' gets written to the port A or D. The ports connect the PDCPU with other parts of the FPGA and the AD9910 DDS.
The bits of port A have the following meaning. Basically they consist of override bits (0, 2, 4) which control whether the PDCPU has access to certain pins of the AD9910 (`OSK`, `DRCTL`, `DRHOLD`) and if so, the level bits (1, 3, 5) control the level of the corresponding pin.

bit(7..6):
: Reserved; write them as 0.

bit(5): drhold
: If bit(4) is set: Level of the DRHOLD pin to the AD9910. If bit(4) is cleared: No meaning.

bit(4): drhold_ovr
: If set, the bit(5) of port A of the PDCPU controls the DRHOLD pin of the AD9910. If cleared (default), the DRHOLD pin is controlled by the registers `DRHOLD_CTLA` and `DRHOLD_CTLB`.

bit(3): drctl
: If bit(2) is set: Level of the DRCTL pin to the AD9910. If bit(2) is cleared: No meaning.

bit(2): drctl_ovr
: If set, the bit(3) of port A of the PDCPU controls the DRCTL pin of the AD9910. If cleared (default), the DRCTL pin is controlled by the registers `DRCTL_CTLA` and `DRCTL_CTLB`.

bit(1): osk
: If bit(0) is set: Level of the OSK pin to the AD9910. If bit(0) is cleared: No meaning.

bit(0): osk_ovr
: If set, the bit(1) of port A of the PDCPU controls the OSK pin of the AD9910. If cleared (default), the OSK pin is controlled by the registers `OSK_CTLA` and `OSK_CTLB`.

The bits of port D control the 3 profile select signals into the AD9910 which allow to select one of 8 profiles:

bit(7..4):
: Reserved; write them as 0.

bit(3..1): profile
: Select one of 8 profiles from the AD9910. The bits 3 to 1 correspond to the `PROFILE2` to `PROFILE0` pins of the AD9910 and allow for very fast change of parameters.

bit(0): enable
: This bit must be written as 1 in order for the bits 3...1 to be effective. If this bit is written as 0, nothing happens.

For port D it is important to understand that both port D and the FPGA registers `PROFILE_CTLA` and `PROFILE_CTLB` share concurrent access to the DDS profile selection. This is unlike port A where the override

bits select whether the PDCPU port A or the FPGA control register accesses the resources of the AD9910 DDS.
The port D allows to change profiles in any order, not only via a Gray code counter. Although you can make any value transition at the profile via port D, it is recommended to use transitions where only one bit changes at a time (like e.g. Gray code/counter).

CPU Instruction `COND`

This is a conditional instruction.
If the condition is true, skip the next instruction.
The `COND` instruction allows to construct if/else branching and loops which are exeucted until a certain externally set condition is true. This can be accomplished by placing `JUMP` instructions directly after the `COND` instruction.
For example to have instructions A,B,C executed when the condition is true and D,E executed when the instruction is false, do the following:

```
   Adr  Instruction
 -------------------------------------------------------------------------------
   100  COND ...         // when condition is true, next instruction is skipped
   101  JUMP 106         // when condition is false, jump to 106
   102  <A>              // enter here only if condition is true
   103  <B>
   104  <C>
   105  JUMP 108         // skip the "else" branch
   106  <D>
   107  <E>
   108  ...
```

The `COND` instruction has the form:

```
  00 Irrrrrrr 00010001   CONDition
```

The logic behind the condition is the same as in the case of the `WAIT FOR EVENT` instruction described above (see `WAIT`): The 7 'r' bits make up a bitmask to select events and the 'I' bit allows to logically invert the end result.

CPU Instruction `SPI` (v2 only)

The `SPI` instruction gives the PDCPUv2 access to the serial configuration interface into the AD9910. This allows the PDCPU to change the register contents of the AD9910. One important application is to implement a series of frequency ramps/sweeps which are completely self-timed by the FlexDDS slot without the requirement to provide an external trigger for each individual ramp.
The `SPI` instruction has the following form:

```
    00 dddddddd L.010101   SPI: Transfer data d over SPI into AD9910 (v2 only)
```

The bit `L` is reserved and should always be written as 0.
The eight data bits `d` specify a byte to transfer via the SPI into the AD9910. In order to write a 64 bit register (like the single tone profile register), 9 `SPI` instructions are needed: The first instruction specifies the register address, the next 8 instructions specify the 8x8=64 bits register content. This is just as the AD9910 accepts the data and much like the register writes carried out over the backplane bus from the computer interface.

NOTE: The SPI instructions have unconditional precedence over SPI writes into the AD9910 DDS carried out over the backplane bus. The user has to ensure that SPI writes over the backplane and carried out via the PDCPU do not collide (i.e. a re not performed at the same time). Failure to do so will cause corrupt register contents.

The SPI instruction only takes one cycle (32ns) to execute and merely enqueues the passed 8 data bits d into a SPI FIFO buffer. This SPI FIFO buffer can hold up to 32 bytes. As soon as one byte is put into the SPI FIFO, the FPGA starts to write the content of the SPI FIFO via the serial interface (SPI) into the AD9910 at a rate of 62.5 MHz (16 ns bit cycle time) which is close to the allowed speed limit for the AD9910.

NOTE: In order to write an AD9910 register via the SPI instruction, the register address and the register content must be sent using multiple SPI instructions immediately following each other. Do not execute other instructions in between.

The special instruction WAIT SPI (see WAIT above) allows to make the CPU wait until the FIFO buffer for the SPI register writes(s) has been completely transferred into the AD9910. The user must use either a WAIT SPI or a long enough WAIT DELAY to ensure the register contents have been transferred into the AD9910 DDS core before executing an UPDATE.

NOTE: The FIFO buffer for the SPI access can hold up to 32 bytes including register address and data bytes. This allows to enqueue several register writes. The user must ensure that the buffer does not overflow, otherwise SPI commands are ignored leading to corrupt or missing register contents.

For example, the 32 byte FIFO can hold 3 register writes to 64 bit registers making up $3 \times (8+1) = 27$ bytes plus one 32 bit register write adding 5 bytes (total: 32 bytes). Alternatively, it can hold 6 register writes to 32 bit registers. If more registers have to be written, the user must use WAIT SPI or a sufficiently long WAIT DELAY instructions between register writes.

NOTE: WAIT instructions which are intended to wait for transmission of SPI FIFO buffer data into the AD9910 *must* be performed between *complete* register writes. I.e. the next SPI instruction after a WAIT must be a register address. This is because an I/O reset is performed for the SPI interface of the AD9910 as soon as the SPI FIFO is empty.

The recommended way to write AD9910 register contents via the SPI instruction is:

1. Perform one or more *complete* register writes using SPI instructions followed directly by each other for a maximum of 32 consecutive instructions.
2. Next, wait for the data to be transferred into the SPI using either a WAIT SPI or a sufficiently long WAIT DELAY instruction.
3. Optionally continue at 1. to write more register contents.
4. Wait or do other reconfiguration things.
5. Perform an UPDATE into the AD9910 (either externally or by use of the UPDATE instruction).

CPU Instruction WAIT_XREG (v2 only)

The WAIT_XREG instruction has the form:

```
00 ....wwwe ..011001   Configure extended wait predivider register.
```

Executing this instruction allows to further extend the wait time for the next WAIT DELAY (short or long) instruction: Without WAIT_XREG the maximum delay is a WAIT DELAY LONG(255) which waits for 65284 cycles (about 2 ms). With a preceding WAIT_XREG instruction, this delay can be multiplied with a factor of up to 2^{16} to extend the maximum wait time to more than 2 minutes.

The WAIT_XREG instruction sets a clock divider for the next WAIT DELAY instruction to a value specified by the three w bits if the enable bit e is set to 1. The clock divider divides the cycle clock down by a factor of

$2^{(2w+2)}$ and introduces an additional delay of $2^{(2w+1)}$. The clock divider is reset automatically (to wwwe=0000) by the next WAIT instruction. Hence, the WAIT_XREG must be called before each WAIT instruction which requires extended delay time.

See the table at the WAIT instruction for a summary of timings.

Example: This is one method to wait 33.4 msec:

```
  00 ....0011 ..011001   WAIT_XREG with enable=1, w=001, (divider 2^4)
  00 11111111 01000101   WAIT DELAY LONG with delay 255
```

Any delay up to more than 2 minutes can be created by a sequence of
1. one WAIT_XREG followed by one WAIT DELAY LONG or WAIT DELAY SHORT (coarse delay)
2. one WAIT DELAY LONG or WAIT DELAY SHORT (fine delay).
Since the WAIT_XREG is automatically reset after the first WAIT DELAY, there is no need to call WAIT_XREG to disable the divider for the second WAIT DELAY.

Setting up the AD9910 for the PDCPU

For the PDCPU to work correctly, the AD9910 has to be configured properly.

This requires that the following bits are set in the CFR2 register of the AD9910: Apart from enabling the SYNC clock and the PDCLK (bits 22 and 11) which should always be turned on, **activate the PDCLK inversion (bit 10), instruct the parallel data assembler to remember previous value (bit 6) and finally enable the parallel data bus (bit 4).** All these steps can be done in a single register write.

Enabling and disabling the parallel data bus can be done at any time, even while the parallel data CPU is running. It merely changes the way the AD9910 interpretes the data present on the parallel data bus (use or ignore).

Disabling and re-enabling the PDCLK while the PDCPU is running to pause execution is not recommended although it will generally work.

Recommended steps

The PDCPU operates largely independent of the rest, so you can use the PDCPU while at the same time modifying other DDS parameters (such as frequency or amplitude) via the regular AD9910 register interface.

Imagine the PDCPU as a "background task" that feeds the programmed values into the parallel data interface of the AD9910.

Here is the recommended way of programming, starting and stopping the PDCPU:

1. Disable the CPU by writing cpudis= 1 in the PDRAM_CTLA register.
2. Store the instructions ("program" for the CPU) in the RAM as explained above.
3. Reset the CPU by writing crst=1 in the PDRAM_CTLA register.
4. Configure the AD9910 to work with the PDCPU (enable parallel data clock; see previous chapter).
5. Start the CPU by writing cpuen= 1 in the PDRAM_CTLA register.
6. (The CPU now runs.)
7. Disable the CPU by writing cpudis= 1 in the PDRAM_CTLA register.

Setting up FlexDDS

USB connector

FlexDDS has a USB connector on the back.

Do not use USB cables longer than 5 meters (limit of USB high speed specification). If no reliable USB connection can be established, try a shorter cable.

RS232 connector

Some versions of FlexDDS have a RS232 connector on the back.

Currently, the RS232 is set up for 115.2kbaud, 1 stop bit, no parity. Hardware handshaking (RTS/CTS) must be enabled, software flow control (Xon/Xoff) must be disabled.

The binary data is the same as for USB. 2 byte words are transferred LSB first.

Be sure to set up the serial port in the host computer to binary so that no CR/LF conversion is performed. Also, character echoing must be disabled.

NOTE: The RS232 internally uses a buffer size of 512 bytes. Only complete buffers will be processed by the rack. Hence, in order to have a buffer processed, fill it up to 512 bytes (256 words) using fill words (0x83 0x00). (This translates into "select no slots for write, do not wait for trigger".) Alternatively, *if this is the last block to transfer*, you might choose 0x03 0x00 ("select no slots for write, wait for trigger"). Filling with 0x00 0x00 produces writes to the CFR1 register in the slots and should be avoided.

Power supply

FlexDDS has a wide input (100 to 230V) power supply and a power switch on the back. Power consumption is 50-60W when all channels are active.

Heat sinking

FlexDDS can be run "as is" when sufficient cool air can passively convect through the case. This requires at least 10cm free space below and above the rack and the air from below should be cold. In space constrained environments it is recommended to install ventillators to increase air throughput. The power supply inside the rack delivers 5.8V and has sufficient strength to deliver up to 1A for ventillators.

FlexDDS RF generator slots

No slot hot plugging! Never ever remove or insert slots from the FlexDDS rack while the power is switched on. Always, switch off the power using the power switch on the back (or by unplugging the power cable) before replacing any slots.

Note also that the slots are electronically identical but precise timing is not completely identical. Therefore, it is strongly recommended to not change the order of the slots.

Frontpanel controls on the slots

TTL in A	BNC input (opto-coupled) and switch (high/low/BNC input). By default, this has no meaning (this can be changed; see the register description for `OSK_CTLA`, `DRCTL_CTLA`, `DRHOLD_CTLA`, `PROFILE_CTLA`).
TTL in B (OSK)	BNC input (opto-coupled) and switch (high/low/BNC input). By default, this is the OSK input which can be used to switch on/off the RF output digitally.
RF level	Potentiometer, switch and BNC input. If the switch is set to external, the "RF level analog in" (0-10V BNC input) can be used to set the RF power level. 0V (disconnected) is maximum level.
Comp. In (10MHz)	Comparator input, not opto-coupled, up to 10MHz. General purpose input with no meaning by default.
Ext Trig	BNC input (opto-coupled). By default, this has no meaning.
Green LED (top)	Power LED attached to microcontroller. Blinks during bootloading and in case of error. Should be lit continuously during normal operation.
Green LED (bottom)	"OK" led. By default shows if the PLL is locked. Should be lit during normal operation.
Red LED	Trigger led; flashes shortly for update triggers to the DDS.
Red pushbutton	Manual trigger button. Mainly useful for testing.
Black pushbutton	Slot reset. Normally, do not use because after reset, certain configuration values need to be stored into the slot. This is only performed if the whole slot is reset.

Frontpanel controls on the left (frontpanel rack control)

Tiny slot on the left; top-to-bottom:

Blue LED	Microcontroller-attached power LED. Blinks in case of error. Should be lit continuously during normal operation.
Yellow LED	USB activitly LED. LED is switched on whenever the rack reads from the USB or wait for additional data from the USB. When the LED is constantly on, more data on the USB needs to be supplied and the trigger is deactivated.
Green LED	Rack trigger LED. Flashes quickly for every trigger event.
Black pushbutton	Rack reset pushbutton.
Red pushbutton	Rack trigger pushbutton.
BNC input (trigger)	Trigger BNC input, opto-coupled. Rising edge triggers.
BNC input (reset)	Rack reset BNC input, opto-coupled. A HIGH level resets the rack.
BNC input (10MHz)	10MHz reference clock input.
Green LED	LED is on if a (10MHz) reference clock is detected.
SMA output	Output: Delivers a synchronized trigger edge for every trigger input. Polarity can be changed via the ICFG register (default: rising edge).
LEMO connector	Used for firmware updates.
Green LED	Power LED.

AD9910 Register Contents: What you must NOT modify!

All registers of the AD9910 DDS chip can be changed by the user. However, some registers and register values should not be changed:

- CFR1: Can be modified; **bits 1 and 0 must always be written as 1 and 0 (SDIO is only input, MSB first)**.
  Defaults: manual external OSK and OSK are enabled, the inverse sinc filter is enabled and SDIO is input only.
- CFR2: Can be modified. **The SYNC clock must always be enabled on the master slot and should be enabled on the slave slots (bit 22 written as 1). The PDCLK clock should be enabled on all slots (bit 11).** By default, singletone ASF, the SYNC clock and the PD clock are enabled and "matched latency" is set.
- CFR3: Do not modify.
- MCS: (multi-chip sync, address 0x0A) Can be modified if the need arises. Ask vendor for instructions and defaults before you do.

Errata

- E1 **Problem:** Setting the Gray counter using `PROFILE_CTLB` does not work reliably with a single write if more than 1 bit changes during the write.

  Workaround: To reliably reset the counter to 0, do not use the cclr bit. Instead, use cload with the following load sequence:

  ```
  PROFILE_CTLB = 00001111    -- load 111
  PROFILE_CTLB = 00001011    -- load 011
  PROFILE_CTLB = 00001001    -- load 001
  PROFILE_CTLB = 00001000    -- load 000
  ```

  Any other value can be loaded in the same fashion: First load the binary complement, and then successively flip the 3 bits, one by one.
  It is recommended to use Gray encoding to switch between different profiles, e.g. binary: 000,001,011,010,110,111,101,100.

  Solution: None.

- E6 **Problem:** A "select slots for write" has to be performed after every "wait for trigger".

  Workaround: Insert "select slots for write" (0x00 0x83) after each "wait for trigger" (i.e. each time the CONTinuation bit is cleared).

  Solution: A firmware update which fixes this bug is available.

Example 1

(FIXME: Not verified for newest version - should work though.)

This is an example how to write into channels 3 and 4 a frequency tuning word for 10MHz, then wait for a trigger. Upon the first trigger, the channel 3 is updated, after the next trigger, channel 4 is updated. 10MHz corresponds to a frequency tuning word (FTW) of 42949673, which is in hex notation: 0x28f5c29, split up in 4 bytes: 0x02, 0x8f, 0x5c, 0x29.

In the example, each line corresponds to a 16bit value interpreted by the DDS rack. The hex and binary columns display the same values in different representations.

```
Hex         Binary
            7      0 15     8
            DDDDDDDD C.....xx  (x = A or R)
-----------------------------------------------------------------------------------
0x18 0x83   00011000 10000011   <-- Select slots 3 and 4 for writing.
0x0e 0x80   00001110 10000000   <-- Write to DDS in slots: AD9910 reg adr 0x0E (STP0)
0x3f 0x80   00111111 10000000      STP0 register: MSB: ampl scale fact high byte = 0x3f
0xff 0x80   11111111 10000000      STP0 register:      ampl scale fact low byte = 0xff
0x00 0x80   00000000 10000000      STP0 register:      phase offset word high byte = 0
0x00 0x80   00000000 10000000      STP0 register:      phase offset word low byte = 0
0x02 0x80   00000010 10000000      STP0 register:      FTW first byte (MSB) = 0x02
0x8f 0x80   10001111 10000000      STP0 register:      FTW next byte = 0x8f
0x5c 0x80   01011100 10000000      STP0 register:      FTW next byte = 0x5c
0x29 0x80   00101001 10000000      STP0 register: LSB: FTW last byte (LSB) = 0x29
   (here, more writes could follow)
0x08 0x05   00001000 00000101   <-- Select channel 3 for trigger, wait for trigger
  ...now the DDS rack waits for a trigger to arrive...
     After the trigger, channel 3 outputs 10MHz.
0x00 0x83   00000000 10000011   <-- according to errata E6
0x10 0x05   00010000 00000101   <-- Select channel 4 for trigger, wait for trigger
  ...now the DDS rack waits for a trigger to arrive...
     After the trigger, channel 4 outputs 10MHz as well.
-----------------------------------------------------------------------------------
```

Just for clarity, the acutal data sent over the USB is simply a character stream with the content from the above dump, read left-to-right, top-to-bottom: In hex notation, the actual character data buffer sent over the USB is:

```
0x18 0x83 0x0e 0x80 0x3f 0x80 ... 0x10 0x05 0x00 0x00 0x00 0x00...
```

(Use enough 0x00 at the end to fill up to the next buffer boundary: 512 bytes for RS232, 1024 for USB.)

Example 2

This is similar to example 1, except that only slot 3 is set to 10MHz and that the slot is updated without the need for an externally generated trigger. Instead, the trigger is generated by sending the appropriate synthetic trigger command over the USB link.

```
Hex         Binary
            7      0 15     8
            DDDDDDDD C.....xx  (x = A or R)
-------------------------------------------------------------------------------------
0x08 0x83   00001000 10000011   <-- Select slot 3 for writing.
0x0e 0x80   00001110 10000000   <-- Start of DDS STP0 write as in example 1.
0x3f 0x80   00111111 10000000
0xff 0x80   11111111 10000000
0x00 0x80   00000000 10000000
0x00 0x80   00000000 10000000
0x02 0x80   00000010 10000000
0x8f 0x80   10001111 10000000
0x5c 0x80   01011100 10000000
0x29 0x80   00101001 10000000   <-- End of DDS STP0 write as in example 1.
0x08 0x85   00001000 10000101   <-- Select slot 3 for trigger but do not wait.
0x01 0x81   00000001 10000001   <-- Send command "generate trigger".
-------------------------------------------------------------------------------------
```