Arduino Due Pulse programmer Documentation

(michal@physics.ubc.ca**)**

Due Pulse Programmer Features:

- Low cost (eg $12)
- Up to 12,000 events
- 25 real-time digital outputs
- 200 ns minimum event time
- 20 ns event granularity
- ~86 s maximum event time
- Looping* (nested, nesting is stack based, can be many levels deep)
- Subroutines* (similarly nested)
- Second digital output port with 24 digital outputs. These can be set either during pulse program execution* or while the programmer is idle.
- Two unipolar 12-bit analog outputs, settable either during program execution* or while the programmer is idle
- 4 analog inputs with 12 bit ADCs, readable while idle.
- 10 MHz square wave output for locking clock to other equipment
- 12 MHz drive clock***
- Trigger input for synchronizing pulse program events to external events. Can wait forever, or be time limited.
- “On the fly” downloading of pulse programs. The next pulse program can be downloaded into the board during the final event (eg FID collection or relaxation delay) of the previous program for perfect timing.
- Fast pulse program downloading: Short programs can be downloaded and restarted in just a few milliseconds (depends on host computer and USB timing) Maximally complicated programs take 25 ms to download.
- Easy expansion. Multiple boards can be easily paralleled for additional real-time outputs and started simultaneously.
- Platform agnostic USB connection.
- Simple to use, C language lightweight API for programming
- C-language API for downloading and execution.
- Clean abort. Sequence may be stopped over USB and outputs set to default values.
- Hardware available from multiple vendors on multiple continents.
- Minimal external hardware required**
- Source code is a single arduino sketch, can be built and flashed from any computer that supports a recent arduino IDE.

* The beginnings and ends of loops and subroutines, as well as setting the outputs of the alternate digital outputs and analog outputs require minimum event durations of ~ 500 ns.
** For “tick-perfect” timing, an external latch chip is required to gate the outputs. If 10 ns fluctuations in output timing can be tolerated, no external latch is necessary.
*** The 12 MHz drive is required for native USB operation. The on-board 12 MHz crystal can be replaced with a 12 MHz signal from another source without any other modifications. A 10 MHz clock

could be supplied instead, however pulse program download would then need to take place over a UART connection, and would be much slower (~20X).

External Connections

1) **Start trigger**: If only a single board will be used, start trigger connections are not necessary, but the **e** command must be used to begin program execution.
If multiple boards are going to used with start-up synchronized, connect pin D2 on the master to D12 on the master and to D12 on all the slaves. All boards should be started with the **E** command. Ensure that the master is started last. For long-term synchronization, the drive clocks of the different boards must also be synchronized. This involves removing the on-board crystals (from at least the slaves) and supplying the 12 MHz clock from another board, or some external source. The clock starts on a rising edge.

2) **Latch:** Connect D11 to the clock input of an external latch chip.

3) **External trigger**: Pin 22 is a digital input for external trigger events. It is **not** 5V tolerant. Both a series resistor (1 kΩ) and a pull-down (2 kΩ) are recommended if the source is 5 V. The input has a weak internal pull-up (~100 kΩ) enabled. Triggers occur on low -> high transitions. For a 3.3 V source, a series resistor is recommended.

4) **10 MHz output**: Pin D53 carries a 3.3V square wave at 10 MHz.

5) **Digital outputs** (For pins, see table below). 3.3 V digital outputs. 15 mA source, 9mA sink current capability on main outputs. Some alternate port pins have reduced (3mA/6mA) current capability – see Table 45-2 in the Sam-3X datasheet.

6) **Analog outputs**: DAC0 and DAC1 pins. I believe these are ~3mA outputs

Pulse program format:

Understanding this section is not required to use the device. A library is provided that handles the generation of pulse programs.

The pulse program that gets downloaded to the Due is a series of blocks. Each block starts with a 32-bit header that has two 16 bit fields. The upper 16 bits represent an opcode (OP) specifying the action taken at the **end** of the block. The lower 16 bits specify the number of events (N) in the block. Only opcodes 0, 1, and 2 can have events (ie N > 0). Following the header, there are N pairs of 32-bit words, where the first word in each pair is the output value and the second word is the number of timer ticks (20 ns) for that event. Following the event words, there are 0 or more arguments for the opcode action.

OPCODE	**Data words**	**Arguments**	**Description**
Any timed event	2 -output word -timer value		

START_LOOP (0)	+2 -loop counter, -new block header	number of iterations	Indicates the start of a loop. Parameter is how many times the loop should be executed
END_LOOP (1)	+1 -new block header		Indicates the end of most recently started loop.
BRANCH (2)	+1 -new block header	header to jump to. In some sense this isn't really an argument, as the header is the start of the next block.	Used to exit from the program and also to link to all of the remaining opcodes.
EXIT (3)	+1 -block header=branch address for exit		When the final event of the pulse program has been started, this is used to exit.
SUB_START (4)	+1 -block header		Indicates the start of a subroutine. This is a slightly weird one because the opcode is used in the downloaded pulse program in the header of the subroutine itself. But it is never used there. It is copied, just before download, as an argument to the CALL SUB opcode.
SUB_END (5)	+1 -new block header		Indicates the end of a subroutine.
EXT_TRIG (6)	+1 -new block header		Indicates that pulse program execution should stop and wait for an external trigger pulse.
TRIG_MAX (7)	+1 -new block header		Same as previous, except that the wait for an external trigger pulse is time limited.
WRITE_DACS (8)	+2 -new block header -dac value	32 bit word containing dac vals One of the 16-bit half words should have 1 << 12 set to set the second dac.	Used to set the DAC output values. This sneaks in after the write to the active port.
WRITE_ALT (9)	+2 -new block header -output value	32 bit output word for alternate port	Used to set the output pins on the alternate output port. This sneaks in after the write to the active port.
SWAP_TO_ALT (10)	+1 -new block header		Used to switch the active output port to the alternate port.
SWAP_TO_DEFAULT (11)	+1 -new block header		Used to switch back so the active output port is the default.
SWAP_TO_DACS (12)	+1 -new block header		Sets the DACS to be the active output port.
WRITE_DEFAULT (13)	+2 -output value -new block header	1) 32 bit output word for default port	Used to write to the default output port when the active port is set to the alternate or the DACS. This is different from WRITE_DACS and WRITE_ALT. It does the port

			write as part of the previous event, so that it is latched on time with the requested event. If not using the latch, this event will be too long.
CALL_SUB (255)	+3 -address for start -data offset -new block header	The header is special, then two args. 0) header: upper 16 bits are 0xffff lower 16 bits are the subroutine id. The subroutine id isn't used by the pulse programmer, but is used before download to calculate the next two parameters. 1) data offset to the start of the subroutine events (calculated in end_code) 2) Block header for the subroutine. end_code inserts this.	Indicates a subroutine call.

Following all of the “normal” blocks, subroutine blocks follow the same pattern with a header, events and arguments. For the subroutines, the headers aren't actually used in place, but get copied in to where the subroutines are called. The final block in each subroutine should have a SUB_END opcode.

Loops can be nested “arbitrarily” deeply, and subroutine calls can be similarly nested. The limit to how deep depends on ram on the arduino. I think there's about 1.3 kB left over when the program is running, so that gives lots of stack room for many levels. Each level takes ~8 bytes.

Once downloaded to the pulse pulse programmer, the block headers are converted into jump addresses, as are any block headers used for subroutine calls, and then the program is ready to be executed.

Program execution

Understanding this section is not required. This is included as a technical reference for how the code operates.

The downloaded program is used to control the execution of the program. The code that executes the program is burned into the Due flash. It is pre-assembled assembly language, and stored as a `const array` called `code[]`. The code consists of several segments. The first segment is essentially an unrolled loop of 12000 events. Each event is 12 bytes of assembly:

```
ldmia r2!, {r1,r4}    ;0xca12    ;load the output word and time value from
                                 ;data, increment data pointer.
wfe                   ;0xbf20    ;wait till timer expires from previous event
str r1,[r0,#56]       ;0x6381    ;write the new output word
str r4,[r5,#28]       ;0x61ec    ;write the new timer value
ldr r1,[r5,#32]       ;0x6a29    ;clear the Rc match flag in the TC_SR
str r7, [r6,#4]       ;0x6077    ;clear the interrupt in the NVIC
```

More on that later.

At the end of that unrolled loop is a short segment of assembly that executes a loop start. Then there is second unrolled loop that executes a loop end, then a third unrolled loop that ends with a few bytes of assembly that branches to the next address provided in the data.

So, now it should be clear why only opcodes 0, 1, 2 can have events. Every other opcode is executed following a branch from opcode 2. In the code, only opcodes 0, 1, and 2 have the code to execute events before the code to execute the opcode.

Each event is executed using the assembly above, where the output word and timer value are loaded into registers r1 and r4. The data pointer, r2, is advanced. Then the cpu is put to sleep with the wfe instruction. When the previous event is over, an *event* (italicized to indicate an arm cpu event) is generated (because the SEVONPEND bit is set in the SCB->SCR (system control block, system control register), and the cpu is awoken. At that point we write the new output word, then the new timer value, then clear the interrupt from the timer's status register and the interrupt controller. In a perfect world, the output word write timing would be perfect, happening as the first operation after the wfe, but it appears as though there is 10 ns of jitter on the write – ie the output word either comes at the right time or 10 ns early. If you want the timing to be perfect, use the latch output and an external latch. The latch output is controlled by the timer, and should be perfect.

Timer

Events are timed by a timer. We use timer 8, aka TC2, channel 2. Somewhat confusing. The timer registers are accessed through TC2->TC_CHANNEL[2]. It is referred to as TC8 for purposes of its interrupt and peripheral clock. The timer is configured in waveform mode. It counts from 0 up to Rc and then resets, apparently Rc resets to 0 in the same tick so that if you set Rc to 10, the period is 10. Timer output TIOA is configured to go low on Rc match, then high on Ra match, 6 ticks later. This is hooked up to pin D.7 (arduino pin D11) and is expected to be used to latch the outputs. The timer is configured to produce interrupts on Rc match as well. During program execution, the timer interrupt priority is insufficient to actually trigger an interrupt, but because of the SEVONPEND setting, it is enough to wake the cpu from a wfe.

Timer start

In order to allow multiple boards to be connected together, the timers can be started by an external trigger. This trigger is received on pin D.8 (arduino pin D12). The signal to start is produced on pin B.25 (arduino D2). So to use multiple boards, connect D2 on one board (the master) to D12 of all the boards, leaving D2 of the slaves unconnected. Then send start instruction to all the slaves first, and finally to the master. This whole system can be bypassed with an alternate start instruction.

Interrupts

The interrupt situation is a bit complicated. The basic idea is that the whole program runs inside an interrupt handler – there are ways this could have been avoided, but they wouldn’t have been any simpler. This is needed because we want to be able to manipulate the interrupt pending flags on the interrupt controller (NVIC). So there are four interrupts that are used.
1) Timer 1 (TC0 channel 1) – this is kind of a dummy interrupt. We configure this timer to produce an interrupt and stop on Rc match. The timer is then started to get execution into the TC1 interrupt handler, which actually executes the pulse program.

2) Timer 8 (TC2 channel 2) – this is triggered at the end of every event. During execution, these interrupts are set to a priority too low to actually trigger, but they wake up the cpu from wfe. The program exits the Timer 1 handler immediately after the final event of the sequence is started then the priority mask changed

so that this interrupt will trigger. When it does trigger, it sets the outputs to default values and forces the latch low then high. In this case, the final event lasts about 380 ns longer than requested.

3) PIOB. This interrupt is used for external trigger events from pin B.26 (arduino pin D22). The pin is set up for interrupts on rising edges, they are enabled only during a wait for trigger event, then immediately disabled. In fact, this interrupt is never enabled on the NVIC, because it doesn't need to be to produce *events* to wake up from WFE.

4) USB interrupts. Our timer 1 interrupt handler, just before beginning pulse program execution, hijacks the USB interrupt vector (ok, there's a real vector set in flash, but the CMSIS library takes that and allows to set a handler with a function call). We record the stored function (from the arduino library), and set out own handler. This is so that pulse program execution can be aborted by sending any character over the USB interrupt. At the end of pulse program execution the arduino handler is restored. Our handler calls the arduino handler in any case so that nothing is lost or confused.

The way the abort works is also a little tricky. When our handler is called, it sets the outputs to defaults, manually flips the latch pin low then high, and then does the tricky stuff. We want to abort execution of the program. To do that, we find the return address that was stored on the stack when the interrupt handler was called, and replace that return address with the address stored as the link address, which is where the pulse program would go when it is ready to exit.

There are a few considerations here. i) To reliably find the return address, we stash the stack pointer in a global variable just before starting the pulse program, then in our handler we can counts down from that stack pointer. A complication here is that stack pointers on interrupt entrance want to by 8-byte aligned. If the stashed stack pointer is not 8 byte aligned, count down an extra 4 bytes.

It seems like it might be simpler to count up from the stack pointer on interrupt entrance, but the stack gets added to twice before we get our hands on it – once at the top of our own handler function, and once inside the CMSIS function that calls it. We can count past these, but it seems dangerous to rely on how many registers get pushed onto the stack by the compiler in these two functions.

ii) when we interrupt our program and simply exit, the stack pointer needs to be in the right place. To do that, we don't ever move the stack pointer. Pulse program execution does use the stack (for looping and subroutines), but we basically shadow the stack pointer (in r9) and use it, so that if an abort happens, the stack pointer is correct.

iii) Interrupts can interrupt in the middle of multiple load/store (ie ldmia). We use those in every event, and don't want to have those interrupted. So interrupt during multiple load/store instructions are disabled. This is the DISMCYCINT bit in SCB->SCR.

5) systick interrupts are used for timekeeping. We disable them so they don't send events. But they are reenabled after program execution so that flashing over the native USB port works.

DACS

The Due has two 12-bit DACs for analog outputs. These can be controlled during pulse program execution or while the program is idle. In either case, you provide a single 32bit word that sets both dacs. To set DAC0, one half-word is a 12-bit value. The second half-word is a second 12-bit value, but with bit 12 set ie word_val = dac_val0 | (dac_val1 << 16) | (1 << 28).

Alternate output port

We're using PIO port C for our main output port. Port A has a similar number of output pins available (even more, if we give up UART0). Port A can be set similarly to the DACS – either when the programmer is idle or while its executing.

Clock

The system is configured to run from a 12 MHz clock (apparently this is necessary for the USB interface). A clock PLL system is set to produce a 100 MHz MCK. This is divided by 2 for the timer.
A 10 MHz signal is produced on pin B.14 (PWMH2, arduino pin D53).

The 12 MHz crystal can be removed and replaced with a square wave signal (or a shifted sine). This doesn't need to 12 MHz, could be 10, but then I think USB serial won't work. Would have to use the UART.

For good synchronization between boards, should supply the same clock to all boards.

Assembly code

Understanding this section is not required. This is included as a technical reference for how the code operates.

The basic event code was described above. The code for the other opcodes is:

Loop start:

```
str.w r3, [r9, #-4]!  ;0xf849 0x3d04  ;push previous loop counter onto
                                      ;stack
ldr.w r3, [r2],#4     ;0xf852 0x3b04  ;load new loop counter from data
str.w r2, [r9, #-4]!  ;0xf849 0x2d04  ;save data pointer on stack
ldr.w pc, [r2], #4    ;0xf852 0xfb04  ;jump to execute first block of loop
```

Loop end:

```
subs  r3, #1          ;0x3b01         ;subtract one from the loop counter
beq.n 8020a <cont>    ;0xd003         ;if its zero, branch below
ldr.w r2, [r9]        ;0xf8d9 0x2000  ;load the data pointer from stack
ldr.w pc, [r2], #4    ;0xf852 0xfb04  ;branch back to start of loop
<cont>:
ldmia.w r9!, {r1, r3} ;0xe8b9 0x000a  ;discard saved pointer and load old
                                      ;loop counter.
ldr.w pc, [r2], #4    ;0xf852 0xfb04  ;jump to next block
```

Branch

```
ldr.w pc, [r2],#4     ;0xf852 0xfb04  ;branch to next block
```

Exit

```
bx lr                  ;0x4770          ;return from call into code. Branch
                                        ;to stored address in lr.
```

Subroutine start:

```
ldmia   r2!, {r1, r4} ;0xca12          ;load subroutine args: data offset
                      ;offset and branch address to start subroutine.
str.w r2, [r9, #-4]!  ;0xf849 0x2d04   ;push current data pointer
add     r2, r1        ;0x440a          ;move the data pointer by offset
bx      r4            ;0x4720          ;and branch to run the subroutine
```

Subroutine end:

```
ldr.w r2, [r9], #4    ;0xf859 0x2b04   ;restore data pointer from stack
ldr.w   pc, [r2], #4  ;0xf852 0xfb04   ;jump to next block
```

External Trig:

```
mov r4, #2            ;0xf04f 0402     ;prep to stop the timer
str r4,[r5]           ;0x602c          ;stop the timer
ldr r4,[r8,#76]       ;0xf8d8 0x404c   ;clear the pin interrupt
                                       ;in PIOB->PIO_ISR (clear on read)
mov r1,#0x1000        ;0xf44f 0x5180   ;prep to clear pending on NVIC
                                       ;PIOB is int 12
str r1,[r6]           ;0x6031          ;Clear pending on NVIC->ICPR[0]
mov r4,#0x4000000     ;0xf04f 0x6480   ;enable the interrupt in PIO_IER
str r4, [r8,#0x40]    ;0xf8c8 0x4040
mov r1,#5             ;0xf04f 0x0105   ;prep for restarting timer
wfe                   ;0xbf20          ;wait for the external trigger
str r1, [r5]          ;0x6029          ;restart the timer
str r4, [r8,#0x44]    ;0xf8c8 0x4044   ;turn off the pin interrupt: PIO_IDR
ldr r4, [r8, #76]     ;0xf8d8 0x404c   ;clear the int in PIO_ISR
mov r1,#0x1000        ;0xf44f 0x5180   ;clear pending flag in NVIC->ICPR[0]
str r1, [r6]          ;0x6031
ldr pc, [r2], #4      ;0xf852 0xfb04   ; jump to next block
```

External Trig with timeout:

```
ldr r4,[r8,#76]       ;0xf8d8 0x404c   ;clear the pin interrupt
                                       ;in PIOB->PIO_ISR (clear on read)
                                       ;in PIOB->PIO_ISR (clear on read)
mov r1,#0x1000        ;0xf44f 0x5180   ;prep to clear pending on NVIC
                                       ;PIOB is int 12
str r1,[r6]           ;0x6031          ;Clear pending on NVIC->ICPR[0]
mov r4,#0x4000000     ;0xf04f 0x6480   ;enable the interrupt in PIO_IER
str r4, [r8,#0x40]    ;0xf8c8 0x4040
mov r4,#50            ;0xf04f 0x0432   ;prep to set Rc to 50 = 1us delay
mov r1,#5             ;0xf04f 0x0105   ;prep to re-zero timer
wfe                   ;0xbf20          ;wait for ext. trigger or timer
str r4,[r5,#28]       ;0x61ec          ;set Rc to saved 50
str r1,[r5]           ;0x6029          ;zero the timer:
mov r4,#0x4000000     ;0xf04f 0x6480   ;disable the interrupt in PIO_IDR
str r4, [r8,#0x44]    ;0xf8c8 0x4044

ldr r4,[r8,#76]       ;0xf8d8 0x404c   ;clear the pin interrupt
```

```
                                        ;in PIOB->PIO_ISR (clear on read)
                                        ;in PIOB->PIO_ISR (clear on read)
ldr r4,[r5,#32]       ;0x6a2c           ;clear timer interrupt in TC_ISR
mov r1,#0x1000        ;0xf44f 0x5180    ;prep to clear pending on NVIC
                                        ;PIOB is int 12
str r1,[r6]           ;0x6031           ;Clear pending PIOB int in NVIC->ICPR[0]
str r7,[r6,#4]        ;0x6077           ;Clear pending timer int in NVIC->ICPR[1]
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Write DACS:

```
ldmia r2!,{r1}        ;0xca02           ;load the dac data word
str r1,[r10,#0x20]    ;0xf8ca 0x1020    ;write it to the dac
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Write Alt:

```
ldmia r2!,{r1}        ;0xca02           ;load output word for alt port.
str r1,[r11,#56]      ;0xf8cb 0x1038    ;write the word to the port PIO_ODSR
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Swap to alt:

```
mov     r0, r11       ;4658             ;move the saved alternate port address
                                        ;to the active port register.
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Swap to default:

```
mov     r0, f12       ;0x4660           ;move the saved default port address
                                        ;to the active port register
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Swap to DACS:

```
sub.w   r0, sl, #24   ;0xf1aa 0x0018    ;move the DACC write register into the
                      ;accounting for the different offset of DACC vs PIO
ldr pc, [r2], #4      ;0xf852 0xfb04    ;jump to next block
```

Write default:

```
ldmia   r2!, {r1}     ;0xca02           ;load output word.
str.w   r1, [ip, #56] ;f8cc 0x1038      ;write to the default port
ldr.w   pc, [r2], #4  ;0xf852 0xfb04    ;jump to next block.
```

Registers

r0 points at the active port (PIOC by default).
r1 is a scratch for loading
r2 is our data pointer
r3 is the runtime loop count
r4 scratch for loading.
r5 points to the timer for access to TC_SR and TC_RC
r6 holds pointer to NVIC->ICPR[0]

r7 holds the number needed to clear our IRQ for the timer
r8 holds PIOB, which has the input trigger pin in it.
r9 holds my shadow stack pointer.
r10 points to DACC to write the dac values
r11(aka fp) points to alternate output port (PIOA)
r12 (aka ip) points to default port (PIOC)
r13 = stack pointer (sr)
r14 = link register (lr)
r15 = program counter (pc)

Pins

Function	Arduino pin	SAM 3X pin	
Timer controlled pin for latch	D11	D.7	TC2->TC_CHANNEL[2]: TIOA (TIOA8)
Timer start trigger input	D12	D.8	TIOB8
Output pulse to trigger timer start	D2	B.25	PIOB, bit 25
10 MHz PWM output	D53	B.14	PWMH2
External trigger for pulse program use	D22	B.26	PIOB, bit 26
DAC0 output	DAC0	B.15	
DAC1 output	DAC1	B.16	
Analog Inputs	D62-D65 (A8-A11)	B.17 – B.20	
LED indicator	D13	B.27	
ID0	D25	D.0	
ID1	D26	D.1	
ID2	D27	D.2	
ID3	D28	D.3	

PIOC (default)	arduino pin		PIOA (alternate)	arduino pin	
C.0	-		A.0	D69/CANTX0	
C.1	D33		A.1	D68/CANRX0	
C.2	D34		A.2	A7/D61	
C.3	D35		A.3	A6/D60	
C.4	D36		A.4	A5/D59	
C.5	D37		A.5	-	
C.6	D38		A.6	A4/D58	
C.7	D39		A.7	D31	
C.8	D40		A.8	D0	UART0 RX
C.9	D41		A.9	D1	UART0 TX
C.10	-		A.10	D19	

C.11	-		A.11	D18	
C.12	D51		A.12	D17	
C.13	D50		A.13	D16	
C.14	D49		A.14	D23	
C.15	D48		A.15	D24	
C.16	D47		A.16	A0/D54	
C.17	D46		A.17	D70/SDA1	
C.18	D45		A.18	D71/SCL1	
C.19	D44		A.19	D42	
C.20	-		A.20	D43	
C.21	D9		A.21	D73 no pin	USB TX LED
C.22	D8		A.22	A3/D57	
C.23	D7		A.23	A2/D56	
C.24	D6		A.24	A1/D55	
C.25	D5		A.25	D74/MISO	
C.26	D4 (shared w/ A.29)		A.26	D75/MOSI	
C.27	-		A.27	D76/SCK	
C.28	D3		A.28	D10 (shared w/ C.29)	
C.29	D10 (shared w/ A.28)		A.29	D4 (shared w/ C.26)	
C.30	D72 no pin	USB RX LED	A.30		
C.31	-		A.31		

Currently:
Port C mask of useful pins: 0x37eff3fe – includes pins C.26 and C.29 – has 25 pins
Port A mask of useful pins: 0x0fdffcdf – does not include A.28, A.29, A.8, A.9 – has 24 pins

Keep this arrangement for now since three A pins are only on the SPI connector, and not brought up on shield.

Maximal A mask, using pins D4 and D10, and the UART pins is:
0x3fdfffdf – would give 28 pins! If drop the UART, is 0x3fdffcdf – 26 pins.
In either case, mask for A becomes: 0x13eff3fe – has 23 pins.

Remaining pins:

D.4	D14	B.13	D21
D.5	D15	B.17	A8/D62
D.6	D29	B.18	A9/D63
D.9	D30	B.19	A10/D64
D.10	D32	B.20	A11/D65
B.10	USB	B.21	D52
B.11	USB	B.23	N/C
B.12	D20		

Pulse program API

`int due_init_program(due_prog_t *program, char auto_shift);`
Use to initialize a program structure and prepare it for receiving a program. The `auto_shift` argument is 0 or 1 to specify if you want the output words shifted. The idea is that you can pack all of your output bits into the lowest 25 or 24 bits of the word. If `auto_shift` is selected, these will get shifted into the right places to appear on the default or alternate output ports.

`int due_exit_program(due_prog_t *program);`
Use this to indicate the end of normal program execution. This should be called just before any subroutines are declared.

`int due_finalize_program(due_prog_t *program);`
This is called after all subroutines have been declared. It resolves the subroutine calls. If there are no subroutine calls, this is not necessary.

`int due_add_event(due_prog_t *program, unsigned int outputs, unsigned int ticks);`
Adds a timed event to a the program. If `auto_shift` was selected with the program was initialized, the outputs will be bit shifted. The bit shifting should be smart enough to keep track of when the active output port is swapped. The ticks argument sets the duration of the event in 20 ns ticks. Event times are subject to the minimums in the table below.

```
int due_start_loop(due_prog_t *program, unsigned int loops,unsigned int outputs,
                   unsigned int ticks);
```
Use this to indicate the start of a loop. The loops argument specifies how many times the loop is iterated. The event specified by `outputs` and `ticks` is included in each iteration. Loops can be nested arbitrarily deeply, subject to available ram.

`int due_end_loop(due_prog_t *program, unsigned int outputs, unsigned int ticks);`
Use this to indicate the end of a loop. The event specified by `outputs` and `ticks` is included in each iteration.

`int due_swap_to_alt(due_prog_t *program, unsigned int outputs, unsigned int ticks);`
Switches the active output port to be the alternate port (port A). This event, and all subsequent ones (until another swap_to event is called) will write the output words to the alternate port. This means that any bits that were active in the previous event will stay active.

```
int due_swap_to_default(due_prog_t *program, unsigned int outputs, unsigned int
ticks);
```
Switches the active output port to be the default port (port C). This event, and all subsequent ones (until another swap_to event is called) will write the output words to the default port. The previous active port will stay in whatever state is was set to in the previous event.

```
int due_swap_to_dacs(due_prog_t *program, unsigned int dac0, unsigned int dac1,
                   unsigned int ticks);
```
Switches the active output port to be the dac port. This event, and all subsequent ones (until another swap_to event is called) will write the output words to the dac port. The previous active port will stay in whatever state is was set to in the previous event.

```
int due_call_sub(due_prog_t *program, unsigned int subroutine_id, unsigned int
                    outputs,unsigned int ticks);
```
Use to call a subroutine. The `subroutine_id` is a numeric indicator to specify which subroutine to call. The event specified by outputs and ticks runs, followed by the first event in the subroutine.

```
int due_wait_for_trigger(due_prog_t *program, unsigned int outputs,unsigned int
ticks);
```
Upon reaching this event, the outputs are set as requested, then the timer is stopped and not restarted until a Low -> High transition is detected on the input trigger pin. After the trigger is detected, the timer is restarted, and the requested delay is

activated. There is about 240 ns delay from the trigger time until the timer is again running. This routine will wait for a trigger forever.

```
int due_wait_for_trigger_max(due_prog_t *program, unsigned int outputs,unsigned int
ticks);
```
Upon reaching this event, the outputs are set as requested, the timer is **not** stopped, the event lasts until either the timer expires, or until a Low -> High transition is detected on the input trigger pin. After the trigger is detected, or the timer expires, there is about a 1.12 μs delay till the next event.

```
int due_write_dacs(due_prog_t *program, unsigned int dac0, unsigned int dac1,
                   unsigned int outputs,unsigned int ticks);
```
This will insert a write to the dacs which happens just after the active port is set.

```
int due_write_alt(due_prog_t *program,unsigned int outputs_alt, unsigned int
                   outputs,unsigned int ticks);
```
This will insert a write to the alternate port which happens just after the active port is set. If `auto_shift` was selected, the bits for both `outputs` and `outputs_alt` will be shifted appropriately.

```
int due_write_default(due_prog_t *program, unsigned int outputs_def,
                   unsigned int outputs,unsigned  int ticks);
```
This will insert a write to the default port – this behaves a little differently from the two previous because of the latch used for the default port. The write occurs immediately after the previous event so that it will be latched at the correct time. The output pin will change earlier than it would for a normal write, so of a latch is not being used on the default port, the timing of this event on the default port will be wrong. If `auto_shift` was selected, the bits for both `outputs` and `outputs_def` will be shifted appropriately.

```
int due_start_sub(due_prog_t *program, int subroutine_id);
```
Indicates the start of the subroutine having `subroutine_id`. No event is produced, it just does the set-up for a subroutine to follow.

```
int due_return_from_sub(due_prog_t *program, int outputs, int ticks);
```
This must be the final event of a subroutine.

```
int due_dump_program(due_prog_t *program);
```
This parses the stored program, and dumps it in a somewhat readable format onto the screen.

To use the api, declare one program structure for each board in the system:
`due_prog_t program1, program2;`

Then call `due_init_program(&program,auto_shift)` for each board. Then can mix `due_add_event()`, `due_start_loop()`, `cue_call_sub()`, etc. When the main program is finished, call: `due_exit_program()` to end the program. Then define any subroutines with: `due_start_sub()` and `due_return_from_sub()`. When all is complete, call `due_finalize_program()` to resolve subroutine calls. Then the program is ready to be downloaded and executed.

Serial Interface

Accepts the following commands:

Q query – returns a string: `Due pulse programmer v1`

P set alternate port values. Followed by 4 bytes for the port value. port set immediately. Returns `OK`.

A (for analog) set the dacs. Followed by 4 bytes for the dac values, same format as during program execution. Returns `OK`.

D download pulse program. Followed by 2 bytes specifying the length (in 32-bit words) of the pulse program data. Send all three bytes in a single write for reliable behaviour. Returns: `<size> size ok` or `too big`. The `<size>` is a text representation of the size received. If size ok, then follow with the data. The data should come in 512 byte chunks. After each chunk, returns the number of bytes received so far. Afterwards get `<ch1> <ch2> data received` or `data incomplete`. `<ch1> <ch2>` are checksums calculated from the data.

E and **e** execute the program. If the timer was still running from a previous program, returns: `Use R for restar`t. Otherwise returns `Starting`. Then, it will either return `Was interrupted` or `Final Event started.` If there was no program to execute, it returns: `no program`.

The difference between **E** and **e** is that **E** enables the timer, and sets it to be started with the trigger (useful if there are multiple boards to be synchronized. **e** starts the timer immediately.

R restart the program – to be used when the final event of the previous program is still running. Returns `Restarting` if all is well, or `Too late` if not. Then returns `Was interrupte`d or `Final Event started` like **E** and **e**. Will also return `no program` if there is no valid program downloaded like **E** and **e**.

S request status. Returns one of:
`Status stopped` if the programmer has not been started since reset and after an abort or timeout has been reported
`status executing - can't happen!` if the programmer status is executing
`status final event: %i ticks remain` if the timer is still running for the final event.
`status final_timeout` if the final event timed out.

I Request ID. Reads the four ID pins (D.0 – D.3 = D25-D28) and returns the value in decimal ASCII.

API for talking to the programmer:

```
int due_open_prog(char *device);
```
pass it a device node, eg. /dev/ttyACM0, it will open it and send a Q, makes sure it gets the correct response and returns the file descriptor, or a -1 on error. It returns a file descriptor that gets passed to all the other functions. It also locks it so nobody else can talk to the device at the same time.

```
int due_close_prog(int fd);
```
pass it the file descriptor you got from `due_open_prog`. It unlocks and closes it.

```
int due_download_prog(int fd,due_prog_t *program);
```
download the program to the programmer. returns 0 on success, -1 on error.

```
int due_run_program(int fd, char start_command);
```
start the program with e E or R. Returns 0 on success. -1 on error

`int due_wait_for_completion(int fd, int timeout);`
wait up to `timeout` (in 1/10 of seconds) for the program to either be interrupted or start its final event. returns 0 for normal completion, 1 if the routine times out, and 2 if the sequence is aborted. -1 for an error. If timeout is 0, will wait forever.

`int due_interrupt_program(int fd);`
sends a K to abort the program. Returns `Was interrupted` or `Got K` if it wasn't running.

`int due_write_dacs_now(int fd, unsigned int dac0,unsigned int dac1);`
writes the two dac values while either idle or while final event is running.

`int due_write_alt_now(int fd, unsigned int output);`
Writes the alternate output port either while idle or while final event is running.

`int due_read_analog(int fd, unsigned char pin);`
Read analog value from the specified pin. Pin should be one of 62-65 for A8-A11.

Download speed: over USB, about 0.264 µs/byte. To fill the ram with a maximal length program takes about 25 ms. This is achieved by bypassing the arduino core USB code. Using the arduino code it was much slower, about 5.6 µs/byte, or more than 0.5 s to fill the ram.

Minimum event lengths:

Event Type	Minimum # of ticks	
ordinary event	10 (maybe 9). 8 is flakey.	
Event before start_loop	20 (18?) 16 fails	
end_loop	20 16 fails	
call_sub	25 (24?) 17 fails	
return_from_sub	25 (20 fails)	
Event before: swap_to_alt, swap_to_default, swap_to_dacs	25 (20 fails)	
Event before wait_for_trigger wait_for_trigger	25 (?) 20 (16 fails)	The requested delay **starts** about 240 ns after the external trigger is received.
Event before wait_for_trigger_max wait_for_trigger_max	15 (?) 25	The requested time is a maximum. If this time is reached, it is as if the external trigger was received at that point. There is an additional 1.12 µs delay following the trigger/timeout before the next event.
write_dacs, write_alt	25 (19 fails)	
Event before write_default	20 (18 fails)	
Event before exit_program	25	