08: MCU datasheets ct’d
Project

Brainstorm/propose projects on Ed thread

Project matching form will release early next week

Project must:

- Use PWM, ADC, or DAC
- Have at least one interrupt service routine
- Have a watchdog timer (doesn’t count as your ISR)
- Use at least one of: Serial communication, Wifi, Timer/counter
Why we’re thinking about the project so early

Time to find you resources
Time to refine the design
Time to order supplies
Time for embedded SE process
Skills in upcoming labs

don't call yourself a "gamer"

if you haven't played this game!
Incredible work on your projects, CS 1600! We want to brag about you!
- Arun, Jason, Stephen, and Prof. Zizyte
Keywords for sharing data

**static**

Value of local variable will persist between function calls (is in memory rather than the stack)
Useful in a function like loop() when you don’t want to declare a global variable
Still local to the function

**volatile**

Means variable can change outside of main execution (e.g. by an ISR)
**Always use volatile when working with variables that change in ISRs!**
Tells compiler not to make certain optimizations (never keep value in a register)
Goal of “bare metal programming” is to configure bits inside of the peripheral registers (blue), which directly control the hardware (green).
Dependencies for using DAC

35.5.1 I/O Lines

Using the DAC Controller’s I/O lines requires the I/O pins to be configured using the port configuration (PORT).

Related Links
23. PORT - I/O Pin Controller

35.5.3 Clocks

The DAC bus clock (CLK_DAC_APB) can be enabled and disabled by the Power Manager, and the default state of CLK_DAC_APB can be found in the Peripheral Clock Masking section.

A generic clock (GCLK_DAC) is required to clock the DAC Controller. This clock must be configured and enabled in the Generic Clock Controller before using the DAC Controller. Refer to GCLK - Generic Clock Controller for details.

This generic clock is asynchronous to the bus clock (CLK_DAC_APB). Due to this asynchronicity, writes to certain registers will require synchronization between the clock domains. Refer to 35.6.7 Synchronization for further details.

Related Links
16.6.2.6 Peripheral Clock Masking
15. GCLK - Generic Clock Controller
## Configuring pin in PORT for DAC

### 23.6.3.1 Pin Configurations Summary

**Table 23-2. Pin Configurations Summary**

<table>
<thead>
<tr>
<th>DIR</th>
<th>INEN</th>
<th>PULLEN</th>
<th>OUT</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Reset or analog I/O: all digital disabled</td>
</tr>
</tbody>
</table>
Multiplexing

- I/O on MCU comes in many varieties: GPIO, DAC, ADC, PWM, interrupts, timers/counters, communication
- Limited # of pins on device – pins can be configured to have one of multiple purposes
- **Multiplexing** is the word for selecting this purpose
<table>
<thead>
<tr>
<th>Pin(1)</th>
<th>I/O Pin</th>
<th>Supply</th>
<th>A</th>
<th>B(2)(3)</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMD2xE</td>
<td>SAMD2xG</td>
<td>SAMD2xJ</td>
<td>EIC</td>
<td>REF</td>
<td>ADC</td>
<td>AC</td>
<td>PTC</td>
<td>DAC</td>
<td>SERCOM(2)(3)</td>
<td>SERCOM-ALT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>PA00</td>
<td>VDDANA</td>
<td>EXTINT[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>PA01</td>
<td>VDDANA</td>
<td>EXTINT[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>PA02</td>
<td>VDDANA</td>
<td>EXTINT[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) Pin column indicates the physical pin number.
(2) B column indicates the pin functionality.
(3) DAC column indicates the digital audio output.
(4) TC column indicates the timer counter.
PMUX register in PORT

23.8.12 Peripheral Multiplexing n

Name: PMUX
Offset: 0x30 + n*0x01 [n=0..15]
Reset: 0x00
Property: PAC Write-Protection

There are up to 16 Peripheral Multiplexing registers in each group, one for every set of two subsequent I/O lines. The n denotes the number of the set of I/O lines.

Bit 7 6 5 4 3 2 1 0
Access RW RW RW RW RW RW RW RW
Reset 0 0 0 0 0 0 0 0

The roles/names of bits inside the register.
Values for named bits

**Bits 3:0 – PMUXE[3:0]** Peripheral Multiplexing for Even-Numbered Pin
These bits select the peripheral function for even-numbered pins (2*n) of a PORT group, if the corresponding PINCFGy.PMUXEN bit is '1'.
Not all possible values for this selection may be valid. For more details, refer to the *I/O Multiplexing and Considerations.*

<table>
<thead>
<tr>
<th>PMUXE[3:0]</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>A</td>
<td>Peripheral function A selected</td>
</tr>
<tr>
<td>0x1</td>
<td>B</td>
<td>Peripheral function B selected</td>
</tr>
<tr>
<td>0x2</td>
<td>C</td>
<td>Peripheral function C selected</td>
</tr>
<tr>
<td>0x3</td>
<td>D</td>
<td>Peripheral function D selected</td>
</tr>
<tr>
<td>0x4</td>
<td>E</td>
<td>Peripheral function E selected</td>
</tr>
<tr>
<td>0x5</td>
<td>F</td>
<td>Peripheral function F selected</td>
</tr>
<tr>
<td>0x6</td>
<td>G</td>
<td>Peripheral function G selected</td>
</tr>
<tr>
<td>0x7</td>
<td>H</td>
<td>Peripheral function H selected</td>
</tr>
<tr>
<td>0x8</td>
<td>I</td>
<td>Peripheral function I selected</td>
</tr>
</tbody>
</table>
35.6 Functional Description

35.6.1 Principle of Operation
The DAC converts the digital value located in the Data register (DATA) into an analog voltage on the DAC output (VOUT).

A conversion is started when new data is written to the Data register. The resulting voltage is available on the DAC output after the conversion time. A conversion can also be started by input events from the Event System.

35.6.2 Basic Operation

35.6.2.1 Initialization
The following registers are enable-protected, meaning they can only be written when the DAC is disabled (CTRLA.ENABLE is zero):

- Control B register (CTRLB)
- Event Control register (EVCTRL)

Enable-protection is denoted by the Enable-Protected property in the register description.

Before enabling the DAC, it must be configured by selecting the voltage reference using the Reference Selection bits in the Control B register (CTRLB.REFSEL).
35.6.2.2 Enabling, Disabling and Resetting
The DAC Controller is enabled by writing a '1' to the Enable bit in the Control A register (CTRLA.ENABLE). The DAC Controller is disabled by writing a '0' to CTRLA.ENABLE.

The DAC Controller is reset by writing a '1' to the Software Reset bit in the Control A register (CTRLA.SWRST). All registers in the DAC will be reset to their initial state, and the DAC Controller will be disabled. Refer to the CTRLA register for details.

35.6.2.3 Enabling the Output Buffer
To enable the DAC output on the $V_{OUT}$ pin, the output driver must be enabled by writing a one to the External Output Enable bit in the Control B register (CTRLB.EOEN).

The DAC output buffer provides a high-drive-strength output, and is capable of driving both resistive and capacitive loads. To minimize power consumption, the output buffer should be enabled only when external output is needed.

35.6.2.4 Digital to Analog Conversion
The DAC converts a digital value (stored in the DATA register) into an analog voltage. The conversion range is between GND and the selected DAC voltage reference. The default voltage reference is the internal reference voltage. Other voltage reference options are the analog supply voltage ($V_{DDA}$) and the external voltage reference ($V_{REF}$). The voltage reference is selected by writing to the Reference Selection bits in the Control B register (CTRLB.REFSEL).

The output voltage from the DAC can be calculated using the following formula:

$$V_{OUT} = \frac{DATA}{0x3FF} \cdot V_{REF}$$

A new conversion starts as soon as a new value is loaded into DATA. DATA can either be loaded via the APB bus during a CPU write operation, using DMA, or from the DATABUF register when a START event occurs. Refer to 35.6.5 Events for details. As there is no automatic indication that a conversion is done, the sampling period must be greater than or equal to the specified conversion time.
What this means

Setup:
- Select reference (CTRLB.REFSEL)
- Enable (CTRLA.ENABLE)

Operation:
- Write 10-bit value to DATA

\[ V_{OUT} = \frac{DATA}{0x3FF} \cdot V_{REF} \]
Note: I misspoke about this in class.
We need to use VDDANA, which is the supply voltage (3.3 V) that powers the chip, NOT INTREF.

35.8.2 Control B

Name:CTRLB
Offset:0x01
Reset:0x00
Property:PAC Write-Protection, Enable-Protected

![Control B Diagram]

**Bits 7:6 – REFSEL[1:0]** Reference Selection
This bit field selects the Reference Voltage for the DAC.

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>INTREF</td>
<td>Internal voltage reference</td>
</tr>
<tr>
<td>0x1</td>
<td>VDDANA</td>
<td>Analog voltage supply</td>
</tr>
<tr>
<td>0x2</td>
<td>VREFA</td>
<td>External reference</td>
</tr>
<tr>
<td>0x3</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
35.8.1 Control A

Name:CTRLA
Offset:0x00
Reset:0x00
Property:PAC Write-Protection, Write-Synchronized

Bit 1 – ENABLE Enable DAC Controller
Due to synchronization there is delay from writing CTRLA.ENABLE until the peripheral is enabled/disabled. The value written to CTRLA.ENABLE will read back immediately and the corresponding bit in the Synchronization Busy register (SYNCBUSY.ENABLE) will be set. SYNCBUSY.ENABLE will be cleared when the operation is complete.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The peripheral is disabled or being disabled.</td>
</tr>
<tr>
<td>1</td>
<td>The peripheral is enabled or being enabled.</td>
</tr>
</tbody>
</table>
### 35.8.8 Data DAC

**Name:** DATA  
**Offset:** 0x08  
**Reset:** 0x0000  
**Property:** PAC Write-Protection, Write-Synchronized

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>Reset</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>Reset</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bits 15:0 – DATA[15:0]** Data value to be converted  
DATA register contains the 10-bit value that is converted to a voltage by the DAC. The adjustment of these 10 bits within the 16-bit register is controlled by CTRLB.LEFTADJ.

**Table 35-1. Valid Data Bits**

<table>
<thead>
<tr>
<th>CTRLB.LEFTADJ</th>
<th>DATA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DATA[9:0]</td>
<td>Right adjusted, 10-bits</td>
</tr>
<tr>
<td>1</td>
<td>DATA[15:6]</td>
<td>Left adjusted, 10-bits</td>
</tr>
</tbody>
</table>
Final to-do list:

- Configure dependencies
  - PORT: select Multiplexer function B for PA02
  - GCLK: configure GCLK_DAC
- Configure DAC peripheral
  - Select reference (CTRLB.REFSEL)
  - Enable (CTRLA.ENABLE)
- Operate DAC peripheral
Using the header files

The header file for each peripheral has definitions that begin with `[PERIPHERAL NAME]_[REGISTER NAME]`. So, if we want to use PORT’s PMUX register, we look for `PORT_PMUX` in `port.h`.
/* ------- PORT_PMUXE : (PORT Offset: 0x30) (R/W 8) GROUP Peripheral Multiplexing n ------- */

typedef union {
    struct {
        uint8_t PMUXE:4; /*< bit: 0..3 Peripheral Multiplexing Even */
        uint8_t PMUXO:4; /*< bit: 4..7 Peripheral Multiplexing Odd */
        bit; /*< Structure used for bit access */
        uint8_t reg; /*< Type used for register access */
    } PORT_PMUXE_Type;
}
#endif /* !(defined(__ASSEMBLY__) || defined(__IAR_SYSTEMS_ASM__)) */

#define PORT_PMUX_OFFSET 0x30 /**< \brief (PORT_PMUX offset) Peripheral Multiplexing n */
#define PORT_PMUX_RESETVALUE 0x00 /**< \brief (PORT_PMUX reset_value) Peripheral Multiplexing n */

#define PORT_PMUXE_Pos 0 /**< \brief (PORT_PMUX) Peripheral Multiplexing Even */
#define PORT_PMUXE_Mask (0x0f << PORT_PMUXE_Pos)

#define PORT_PMUXE(value) ((PORT_PMUXE_Mask & ((value) << PORT_PMUXE_Pos)))
#define PORT_PMUXE_A_Val 0x00 /**< \brief (PORT_PMUX) Peripheral function A selected */
#define PORT_PMUXE_B_Val 0x10 /**< \brief (PORT_PMUX) Peripheral function B selected */
#define PORT_PMUXE_C_Val 0x20 /**< \brief (PORT_PMUX) Peripheral function C selected */
#define PORT_PMUXE_D_Val 0x30 /**< \brief (PORT_PMUX) Peripheral function D selected */
#define PORT_PMUXE_E_Val 0x40 /**< \brief (PORT_PMUX) Peripheral function E selected */
#define PORT_PMUXE_F_Val 0x50 /**< \brief (PORT_PMUX) Peripheral function F selected */
#define PORT_PMUXE_G_Val 0x60 /**< \brief (PORT_PMUX) Peripheral function G selected */
#define PORT_PMUXE_H_Val 0x70 /**< \brief (PORT_PMUX) Peripheral function H selected */

#define PORT_PMUXE_A (PORT_PMUXE_A_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_B (PORT_PMUXE_B_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_C (PORT_PMUXE_C_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_D (PORT_PMUXE_D_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_E (PORT_PMUXE_E_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_F (PORT_PMUXE_F_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_G (PORT_PMUXE_G_Val << PORT_PMUXE_Pos)
#define PORT_PMUXE_H (PORT_PMUXE_H_Val << PORT_PMUXE_Pos)

definition of the bit locations/names within the register

definitions for values PMUXE bits take on: offset of those bits within the register, values that these bits can take on

this mask has 1s in all of the bits of PMUXE and 0s everywhere else. See how we could use it on the next slide

these are the relevant definitions (values shifted to the correct position)
Writing the correct value to PMUX

// Select MUX function B for PA02
PORT->Group[PORTA].PMUX[1].reg = PORT_PMUX_PMUXE_B;

// Note that the above would overwrite the PMUXO bits.
// If we need to keep them (i.e. if we're also configuring PA01),
// we should use bit operations and masking, e.g.
// PORT->Group[PORTA].PMUX[1].reg &= ~PORT_PMUX_PMUXE_Msk
// (To clear the PMUXE bits using a mask)
// PORT->Group[PORTA].PMUX[1].reg |= PORT_PMUX_PMUXE_B
// (To set the value of those bits without affecting the other bits)
In dac.h:

Note that the name doesn't exactly match up with the datasheet (VDDANA), but the value the header file defines (0x1) does. This is probably for compatibility reasons with related boards – as long as we've double-checked that we match the datasheet, we can use this

110  #define DAC_CTRLB_REFSEL_Msk  (0x3u << DAC_CTRLB_REFSEL_Pos)
111  #define DAC_CTRLB_REFSEL(value) ( ((DAC_CTRLB_REFSEL_Msk & ((value) << DAC_CTRLB_REFSEL_Pos)))
112  #define DAC_CTRLB_REFSEL_INT1V_Val  0x0u  /**< (DAC_CTRLB) Internal 1.0V reference */
113  #define DAC_CTRLB_REFSEL_AVCC_Val  0x1u  /**< (DAC_CTRLB) AVCC */
114  #define DAC_CTRLB_REFSEL_VREFP_Val  0x2u  /**< (DAC_CTRLB) External reference */
115  #define DAC_CTRLB_REFSEL_INT1V (DAC_CTRLB_REFSEL_INT1V_Val  << DAC_CTRLB_REFSEL_Pos)
116  #define DAC_CTRLB_REFSEL_AVCC (DAC_CTRLB_REFSEL_AVCC_Val  << DAC_CTRLB_REFSEL_Pos)
117  #define DAC_CTRLB_REFSEL_VREFP (DAC_CTRLB_REFSEL_VREFP_Val  << DAC_CTRLB_REFSEL_Pos)
118
119  #define DAC_CTRLA_SWRST_Pos  0  /**< (DAC_CTRLA) Software Reset */
120  #define DAC_CTRLA_SWRST  (0x1u << DAC_CTRLA_SWRST_Pos)
121  #define DAC_CTRLA_ENABLE_Pos  1  /**< (DAC_CTRLA) Enable */
122
123  #define DAC_CTRLA_ENABLE (0x1u << DAC_CTRLA_ENABLE_Pos)
124  #define DAC_CTRLA_RUNSTDY_Pos  2  /**< (DAC_CTRLA) Run in Standby */
125  #define DAC_CTRLA_RUNSTDY (0x1u << DAC_CTRLA_RUNSTDY_Pos)
126  #define DAC_CTRLA_MASK  0x07u  /**< (DAC_CTRLA) MASK Register */
```c
void setup() {
    // ** Configure PORT **
    // Reset config bits for PA02 and enable MUX: TODO (see example in lab 3)
    // Select MUX function B for PA02
    PORT->Group[PORTA].PMUX[1].reg = PORT_PMUX_PMUXE_B;

    // ** Configure GCLK_DAC **
    // TODO (see example in lab 3)

    // ** Configure DAC **
    // Select internal voltage reference
    DAC->CTRLB.reg = DAC_CTRLB_REFSEL_AVCC;
    // Enable DAC
    DAC->CTRLA.reg = DAC_CTRLA_ENABLE;
    // We could have also done this using bit manipulation, e.g.:
    // DAC->CTRLA.reg = (1 << 1); // Move value 1 to bit position 1
}
```
void loop() {
    // Output 2.5V on DAC pin

    // Compute DAC value for 2.5 V
    // (write 25/33 instead of 2.5/3.3 to avoid floating point division)
    unsigned int dac_val = 1023 * 25 / 33;
    // Write to DAC DATA register
    DAC->DATA.reg = dac_val;
}