University of British Columbia
Electrical and Computer Engineering
Electrical and Biomedical Engineering Design Studio
ELEC291/ELEC292

The SAMD20E16 Microcontroller System

Introduction

This document introduces a minimal microcontroller system using the Microchip/Atmel's SAMD20E16 ARM Cortex-M0 microcontroller.

Recommended documentation

SAM D20 Family Data Sheet:
http://ww1.microchip.com/downloads/en/DeviceDoc/60001504B.pdf

SAM D20 Family Silicon Errata and Data Sheet Clarification:
http://ww1.microchip.com/downloads/en/DeviceDoc/80000747B.pdf

Assembling the Microcontroller System

Figure 1 shows the circuit schematic of the SAMD20E16 microcontroller system used in ELEC291/ELEC292. It can be assembled using a bread board. Table 1 below lists the components needed to assemble the circuit. Figure 2 shows the assembled circuit. Figure 3 show the details of the wiring as many wires and resistors are placed under the printed circuit boards.

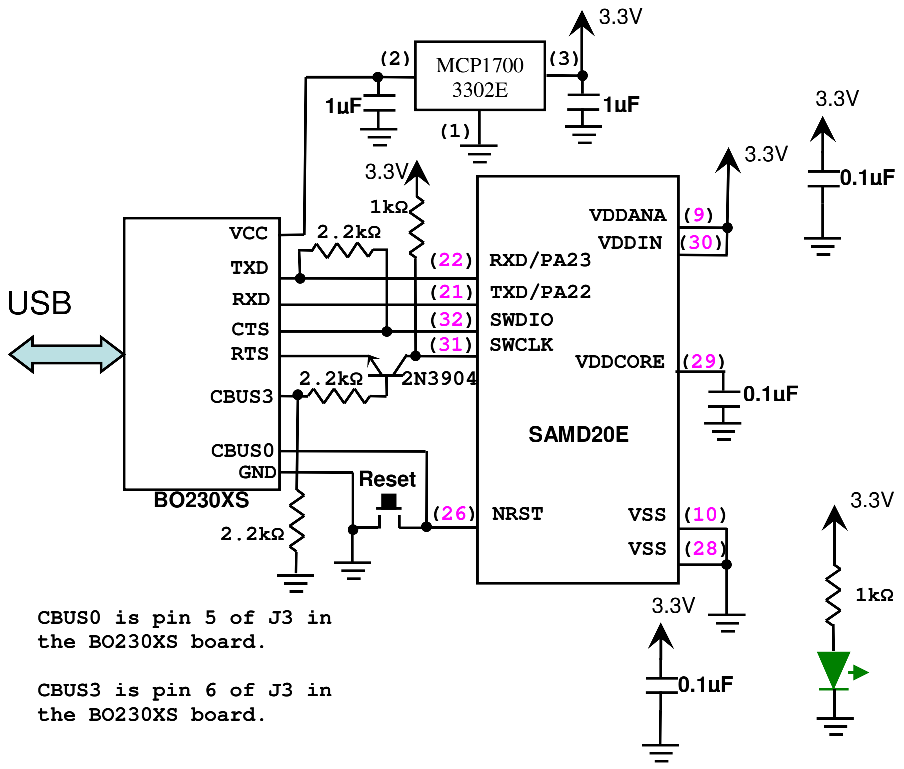
Quantity	Digi-Key Part #	Description
3	BC1148CT-ND	0.1uF ceramic capacitors
2	BC1157CT-ND	1uF ceramic capacitors
2	1.0KQBK-ND	1kΩ resistor
3	2.2kQBK-ND	2.2kΩ resistor
1	67-1108-ND	LED 5MM GREEN
1	MCP1700-3302E/TO-ND	MCP17003302E 3.3 Voltage Regulator
1	N/A	BO230XS USB adapter
1	ATSAMD20E16B-AUTCT-ND	SAMD20E16 microcontroller
1	1528-1065-ND	LQFP32 to DIP32 adapter board
2	A26509-16-ND	16-pin header connector
1	P8070SCT-ND	Push button switch

Table 1. Parts required to assemble the SAMD20E16 microcontroller system.

Before assembling the circuit in the breadboard, the SAMD20E16 microcontroller has to be soldered to the LQFP32 to DIP32 adapter board. This task can be accomplished using a solder iron as described in this video:

https://www.youtube.com/watch?v=8yyUlABj29o

Once the microcontroller is soldered to the adapter board, solder the two 16-pin header connectors. The assembled bread boarded circuit should look similar to the one in figure 1. Notice that the SAMD20E16 works with a VDD voltage of 3.3V. For this reason a voltage regulator is required to convert the USB adapter 5V to 3.3V.



CBUS0 is pin 5 of J3 in the BO230XS board.

CBUS3 is pin 6 of J3 in the BO230XS board.

Figure 1. Circuit schematic of the microcontroller system.

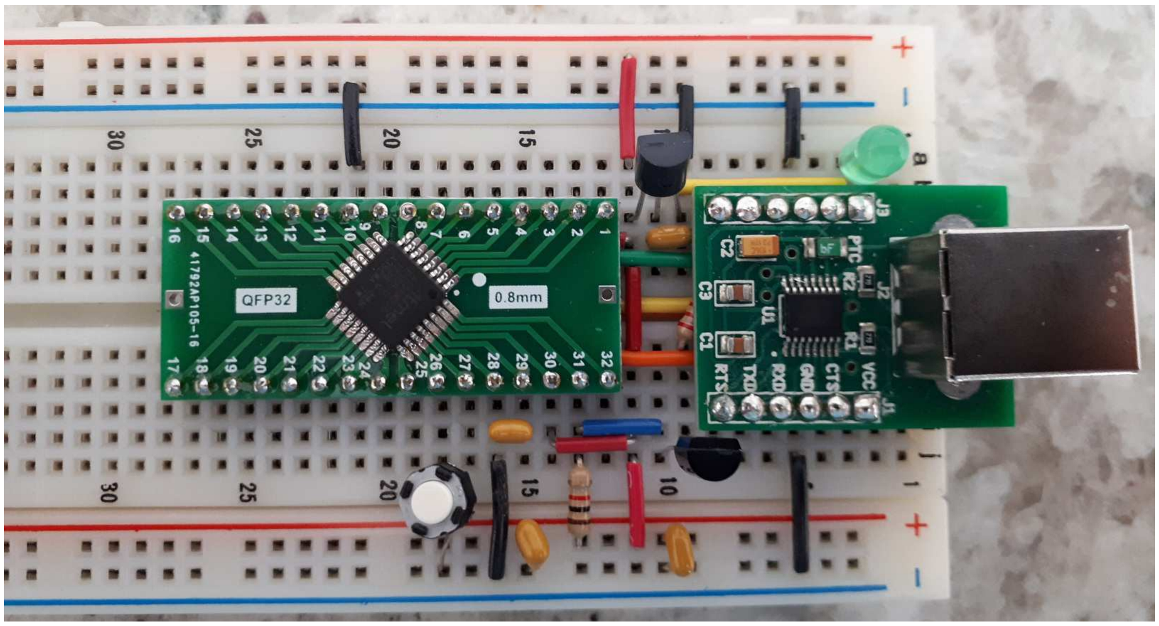


Figure 2. Bread boarded SAND20E16 microcontroller system.

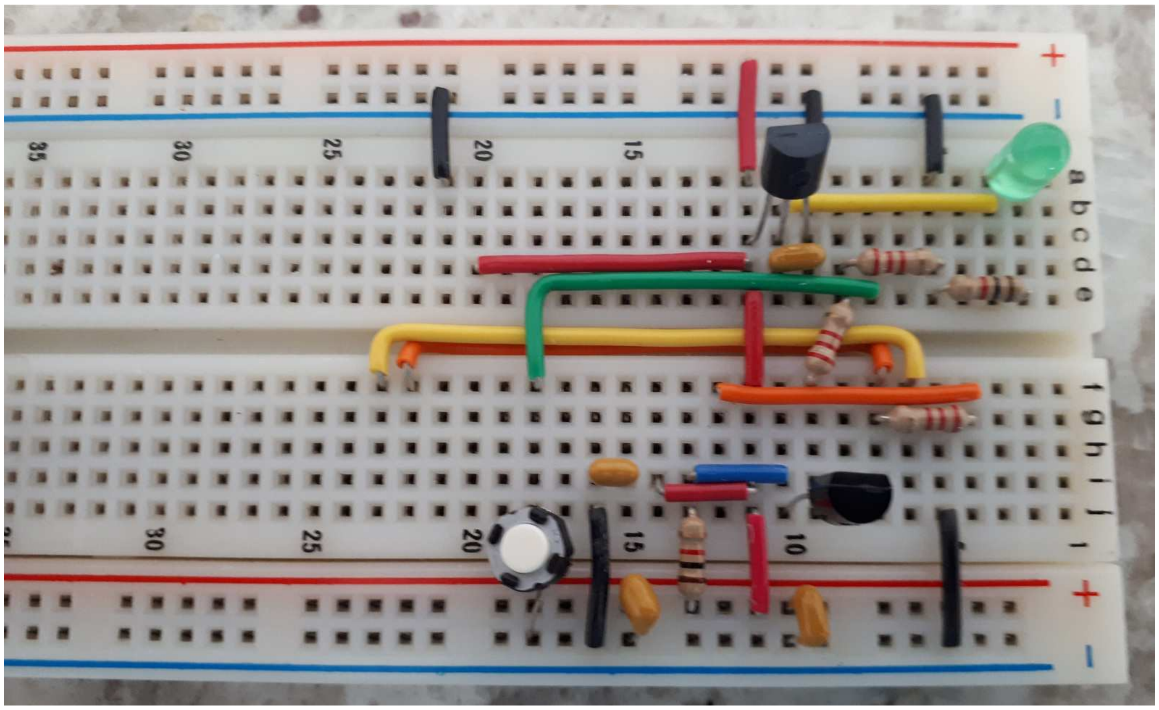


Figure 3. Detailed wiring of the bread boarded SAND20E16 microcontroller system.

Setting up the Development Environment

To establish a workflow for the STM32 we need to install the following three packages:

1. **CrossIDE V2.25 (or newer) & GNU Make V4.2 (or newer)**

   Download CrossIDE from: http://ece.ubc.ca/~jesusc/crosside_setup.exe and install it. Included in the installation folder of CrossIDE is GNU Make V4.2 (make.exe, make.pdf). GNU Make should be available in one of the folders of the PATH environment variable in order for the workflow described bellow to operate properly. For example, suppose that CrossIDE was installed in the folder "C:\crosside"; then the folder "C:\crosside" should be added at the end of the environment variable "PATH" as described here[1].

   Some of the Makefiles used in the examples below may use a "wait" program developed by the author. This program (and its source code) can be downloaded from the course web page and must be copied into the CrossIDE folder or any other folder available in the environment variable "PATH".

2. **GNU ARM Embedded Toolchain.**

   Download and install the GNU ARM Embedded Toolchain from:

   https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads.

   The "bin" folder of the GNU ARM Embedded Toolchain must be added to the environment variable "PATH" in order for the workflow described bellow to operate properly. For example, if the toolchain is installed in the folder "C:\Programs\GNU

[1] http://www.computerhope.com/issues/ch000549.htm

Tools ARM Embedded", then the folder "C:\Programs\GNU Tools ARM Embedded\5.4 2016q2\bin" must be added at the end of the environment variable "PATH" as described here[1].

Notice that the folder "5.4 2016q2" was the folder available at the time of writing this document. For newer versions of the GNU ARM Embedded Toolchain this folder name will change to reflect the installed version.

3. **SAMD20E16 Flash Loader.**

   Available in the web page for the course is the program "Prog_SAMD". The folder of the SAMD Flash programmer must be added to the environment variable "PATH" in order for the workflow described bellow to operate properly. For example, if the "Prog_SAMD" Flash loader is installed in the folder "C:\Programs\Prog_SAMD", then the folder "C:\Programs\Prog_SAMD" must be added at the end of the environment variable "PATH" as described here[1].

Workflow.

The workflow for the SAMD20E16 microcontroller includes the following steps.

1. **Creation and Maintenance of Makefiles.**

CrossIDE version 2.26 or newer supports project management using **simple** Makefiles by means of GNU Make version 4.2 or newer. A CrossIDE project Makefile allows for easy compilation and linking of multiple source files, execution of external commands, source code management, and access to microcontroller flash programming. The typical Makefile is a text file, editable with the CrossIDE editor or any other editor, and looks like this:

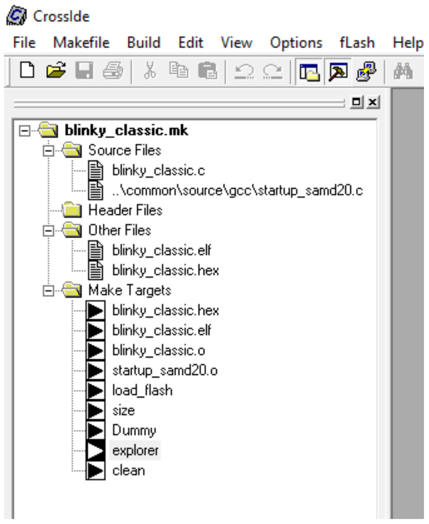
```
# Since we are compiling in windows, select 'cmd' as the default shell.  This
# is important because 'make' will search the path for a linux/unix like shell
# and if it finds it will use it instead.  This is the case when cygwin is
# installed.  That results in commands like 'del' and echo that don't work.
SHELL=cmd
COMPORT=$(shell type COMPORT.inc)
# List the object files to link together
OBJS = blinky_classic.o startup_samd20.o

# Specify the compiler to use
CC = arm-none-eabi-gcc
OBJCOPY = arm-none-eabi-objcopy
SIZE = arm-none-eabi-size

# Compiler options
CFLAGS += -W -Wall --std=gnu99 -Os -fno-diagnostics-show-caret -fdata-sections
CFLAGS += -ffunction-sections
CFLAGS += -funsigned-char -funsigned-bitfields -mcpu=cortex-m0plus -mthumb -MD -MP

# Linker Options
LDFLAGS += -mcpu=cortex-m0plus -mthumb -Wl,--gc-sections
LDFLAGS += -Wl,--script=../common/Source/gcc/samd20e16b.ld

# Place were the samd20.h file is stored
INCLUDES = -I..\common\include

# Definitions related to our design.  In this case the chip we are using
# (SAMD20E16B) and the clock frequency we are using F_CPU=1000000
DEFINES = -D__SAMD20E16B__ -DDONT_USE_CMSIS_INIT -DF_CPU=1000000

CFLAGS += $(INCLUDES) $(DEFINES)

blinky_classic.hex: blinky_classic.elf
	$(OBJCOPY) -O ihex blinky_classic.elf blinky_classic.hex

blinky_classic.elf: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) $(LIBS) -o blinky_classic.elf

# The object file blinky_classic.o depends on blinky_classic.c.
# blinky_classic.c is compiled to create blinky_classic.o.
blinky_classic.o: blinky_classic.c
	$(CC) $(CFLAGS) blinky_classic.c -c -o blinky_classic.o

startup_samd20.o: ..\common\source\gcc\startup_samd20.c
	$(CC) $(CFLAGS) ..\common\source\gcc\startup_samd20.c -c -o
startup_samd20.o

# Target `load_flash ` is used to load the hex file to the microcontroller
# using the flash loader.
load_flash:
	..\..\Prog_SAMD\Prog_SAMD -p blinky_classic.hex

size: blinky_classic.elf
	@echo size:
	@$(SIZE) -t blinky_classic.elf

# Dummy targets can be added to show useful files in the file list of
# CrossIDE or execute arbitrary programs:
Dummy: blinky_classic.hex
	@echo Nothing to see here!

explorer:
	cmd /c start explorer .

clean:
	@echo clean
	del *.o *.d *.elf *.hex
```

The preferred extension used by CrossIDE Makefiles is ".mk". For example, the file above is named "blinky.mk". Makefiles are an industry standard. Information about using and maintaining Makefiles is widely available on the internet. For example, these links show how to create and use simple Makefiles.

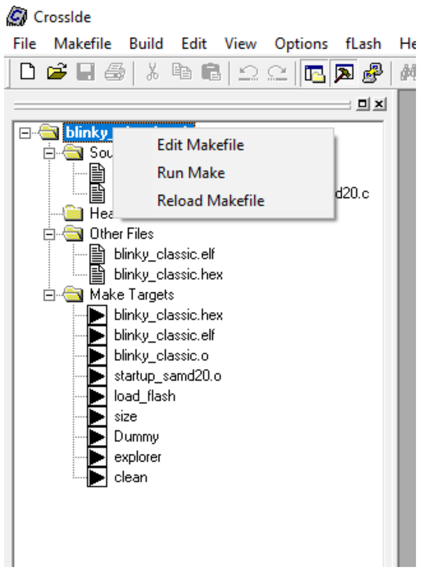
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
https://en.wikipedia.org/wiki/Makefile

2. Using Makefiles with CrossIDE: Compiling, Linking, and Loading.

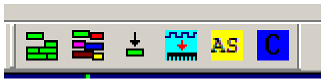
To open a Makefile in CrossIDE, click "Makefile"→"Open" and select the Makefile to open. For example "blinky.mk". The project panel is displayed showing all the targets and source files:



Double clicking a source file will open it in the source code editor of CrossIDE. Double clicking a target 'makes' that target. Right clicking the Makefile name shows a pop-up menu that allows for editing, running, or reloading of the Makefile:



Additionally, the Makefile can be run by means of the Build menu or by using the Build Bar:



Clicking the ‘wall’ with green ‘bricks’ makes only the files that changed since the last build. Clicking the ‘wall’ with colored ‘bricks’ makes all the files. Clicking the ‘brick’ with an arrow, makes only the selected target. You can also use F7 to make only the files that changed since the last build and Ctrl+F7 to make only the selected target.

<u>Compiling & Linking</u>

After clicking the build button this output is displayed in the report panel of CrossIDE:

```
----------------- CrossIde - Running Make -----------------
arm-none-eabi-gcc -W -Wall --std=gnu99 -Os -fno-diagnostics-show-caret -fdata-sections -ffunction-sections -
funsigned-char -funsigned-bitfields -mcpu=cortex-m0plus -mthumb -MD -MP -I..\common\include -D__SAMD20E16B__ -
DDONT_USE_CMSIS_INIT -DF_CPU=1000000  blinky_classic.c -c -o blinky_classic.o
arm-none-eabi-gcc -W -Wall --std=gnu99 -Os -fno-diagnostics-show-caret -fdata-sections -ffunction-sections -
funsigned-char -funsigned-bitfields -mcpu=cortex-m0plus -mthumb -MD -MP -I..\common\include -D__SAMD20E16B__ -
DDONT_USE_CMSIS_INIT -DF_CPU=1000000  ..\common\source\gcc\startup_samd20.c -c -o startup_samd20.o
arm-none-eabi-gcc -mcpu=cortex-m0plus -mthumb -Wl,--gc-sections -Wl,--script=../common/Source/gcc/samd20e16b.ld
blinky_classic.o startup_samd20.o  -o blinky_classic.elf
arm-none-eabi-objcopy -O ihex blinky_classic.elf blinky_classic.hex
```

<u>Loading the Hex File into the Microcontroller’s Flash Memory</u>

To load the program into the microcontroller double click the ‘Flash_Load’ target. This output is then displayed in the report panel of CrossIDE and the program starts running.

```
----------------- CrossIde - Running Make -----------------
..\..\Prog_SAMD\Prog_SAMD -p blinky_classic.hex
SAMD20/SAMD21 SWD programmer using FT230X, FT231X, or FT232R.
(C) Jesus Calvino-Fraga (2016-2019)
Connected to COM12
SWIN<-CTS, SWOUT->TXD, SWCLK->RTS
blinky_classic.hex: Loaded 568 bytes.  Highest address is 567.
SWD IDCODE = 0x0bc11477
DSU DID = 0x1000160c
Serial number= 0xff0f101d472020205150314693f367da
Part Number = SAMD20E16A/B
Erasing device... done
Writing 9 pages to flash memory:
0%    10%   20%   30%   40%   50%   60%   70%   80%   90%   100%
|.....|.....|.....|.....|.....|.....|.....|.....|.....|.....|
############################################################
done (3031.6 bytes/second)
Reading CRC32 from IC... done.
Success! The CRC32 from the IC and the buffer match: 0x68a783d9.
Tasks completed in 0.6 seconds.
```

A file named “COMPORT.inc” is created after running the flash loader program. The file contains the name of the port used to load the program, for example, in thie example above COM118 is stored in the file. “COMPORT.inc” can be used in the Makefile to create a target that starts a PuTTy serial session using the correct serial port:

```
PORTN=$(shell type COMPORT.inc)
.
.
putty:
          @Taskkill /IM putty.exe /F 2>NUL | wait 500
          c:\putty\putty.exe -serial $(PORTN) -sercfg 115200,8,n,1,N -v
```

For more details about using “COMPORT.inc” check the project examples below.

Project Examples

The following Project examples are available in web page of the course. They are all written for the SAMD20E16 ARM microcontroller.

BlinkyClassic: 'blinks' an LED connected to port PA16 (pin 17). This is the same project used in the examples above. A picture is included in the source folder that shows how to connect the LED to the port pin.

BlinkySystick: Similar to "BlinkyClassic" above but uses the Systick counter to implement the time delay.

BlinkyTimerISR: Similar to "BlinkyClassic" but instead of using a delay loop, it uses the timer 3 interrupt.

Printf_Test: Uses printf() and scanf() to read/display strings and variables using the serial port. This example uses the default libraries of GCC which consume a considerable amount of program memory. Do not use this example as a starting point for your own projects, instead use the next example 'Printf_Test_Newlib' which uses a more efficient version of the libraries (Newlib).

Printf_Test_Newlib: Same as above, but uses Newlib instead.

ADC: Reads a channel of the built in ADC (PA05, pin 6) and displays the result using serial port 3 and PuTTY. The ADC reference is provided via PA03 (pin 4). This project uses simple functions to display the ADC output, therefore is very efficient regarding memory usage. A potentiometer connected between 3.3V and GND can be used to test this example, as shown in the attached picture in the source folder.

DAC: Shows how to use the internal Digital to Analog Converter of the SAMD20E16 microcontroller. The output is connected to pin PA02 (pin 3).

LCD_4bit: Shows how to connect, initialize, and use an LCD in 4-bit mode. It uses pins PA00 to PA05 (pins 1 to 6). Notice that the LCD VCC must be connected to +5V, and the LCD R/W signal must be connected to GND. There is a picture of the LCD wiring in the same folder of the example.

LCD_4bit_Ver2: Exactly the same as the example above, but uses different pins to connect to the LCD. There is a picture of the LCD wiring in the same folder of the example.

Pushbutton: Shows how to read a push button connected between PA15 (pin 16) and ground. There is a picture in the source folder that shows how to connect the pushbutton.

Period: Shows how to measure the period of a square wave applied to an arbitrary pin. In this example the pin used is PA14 (pin 15 of LQFP32 package). A 555 timer can be used to test this example. A picture of the wired 555 timer is included in the folder of this example.

SPI_ADC: This example shows how to communicate with the MCP3008 ADC using SPI. SERCOM1 is configured as SPI. Channels 0 and 1 of the MCP3008 are read and converted to voltage and subsequently transmitted via the serial port (using SERCOM3 configured as UART) to PuTTY for display. There is a picture of the wiring between the SAMD20 and the MCP3008 in the folder of the example.

TwoServos: This example produces two standard [0.6 to 2.4 millisecond] pulses every 20 milliseconds which can be used to control a couple of servos. The pulse width can be adjusted via PuTTY and the serial port. The two pulse outputs are arbitrarily set to pins PA16 and PA17. Additionally, pin PA15 is used to check that the Interrupt Service Routine for timer 3 is executed every 10 µs. This is not the most efficient way of controlling servos with the SAMD20, but it is certainly flexible: with one timer ISR an arbitrary number of servos can be controlled using any available pins.

TwoServosTC2TC3: This example produce two standard servo signals to control a couple of servos like the previous example. In this example two timers, together with their predefined pin outputs, are used. Once the timers are set, the pulse width is controlled by writing to one the timer registers. This way the signals are generated without CPU intervention. This is of course more efficient, but only the pins associated with the timers can be used to generate the signals.

SquareWaveTC3ISR: Shows how to configure timer 3 to generate an interrupt where a square wave is produced by toggling an arbitrary port pin, in this case PA16 (pin 17) with a frequency between 100Hz and 500kHz.

SquareWaveTC3CC0: Shows how to configure timer 3 to generate a square wave at port PA14 (pin 15) using the compare registers of the timer and its associated output port in this case PA14.

Uart2: This example demonstrates how to configure SERCOM2 as an UART. It uses ports PA08 (pin 11) for TxD and PA09 (pin 12) for RxD. It also uses Uart3, so there are two UARTs enabled at the same time! For testing you can connect the oscilloscope to PA08 and see how the serial message is transmitted.