~~Mutley's~~ Guide To The Wacky Racers, V.013E-3

Michael Hayes

2021

Contents

Chapter 1

Introduction

This guide is written in the hope that it will ease your introduction into building, programming, and troubleshooting an embedded system. Making embedded systems is not easy; you have to get a lot right. One incorrect bit can be the difference between a working and a non-working system.

If you find things a little Linux-centric, well that's the only operating system I've run for over twenty-five years. It is a common operating system for many high-end embedded systems; it runs on Android phones and probably on your internet router. And, yeah, I like the command-line; it is faster than using a mouse and GUI.

I would like to thank all those who have helped with this assignment over the years: Andrew Bainbrige-Smith, Blair Bonnett, Chris Cameron, Mike Cusdin, Matthew Edwards, Michael Frampton, Jay Gunathailake, Daniel Hopkins, Harry Mander, Daniel Morris, Morgan King, Scott Lloyd, Ben Mitchell, Diego Ramirez, Andre Renaud, Sam Spekreijse, Steve Weddell.

Chapter 2

System design

The system requirements are specified in the assignment instructions.

2.1 Recommendations

1. Independently fuse the buck converter and H-bridge. This allows a fuse to be removed to isolate part of the circuit when finding a fault, such as a short across the power rails.
2. Have a zener diode to protect against overvoltage when your group member inadvertently cranks up the voltage from the bench power supply.
3. Have current limiting resistors for all off-board signals.
4. Have plenty of testpoints, especially for power supplies and signals. You will never have enough!
5. Have at least one grunty ground testpoint for attaching a scope ground clip.
6. Have a dedicated PIO pin to drive a testpoint that you can use to trigger an oscilloscope for debugging.
7. Have the SAM4S erase pin connected to a test point close to a 3.3 V testpoint. This is useful to completely erase the SAM4S flash memory when nothing works.
8. Have a MOSFET or servo interface for controlling something dastardly!

2.2 SAM4S MCU

The SAM4S MCU is overkill for this assignment but is typical of ARM processors used for bare-metal applications. The particular chip we are using is the SAM4S8B. This was made by Atmel and is now owned by Microchip. The datasheet is available at https://www.microchip.com/en-us/product/ATSAM4S8B.

The SAM4S8B has a 32-bit ARM Cortex M4 RISC CPU that can run at 120 MHz and has 512 KB of flash memory and 128 KB of SRAM.

2.2.1 Power pins

The SAM4S has four grounds. They **must** all be connected. There are also seven power pins. These **must** all be connected since they power different parts of the chip. Note, some pins require 3.3 V while others require 1.2 V. The 1.2 V is generated by an internal voltage regulator.

2.2.2 Peripheral pins

Many of the peripheral pins are dedicated and cannot be reassigned in software, e.g., SPI, TWI, and USB pins. Note, there are restrictions on the PWM pins. See Table 11-2 in the SAM4S datasheet for the pin assignments. Also see PIO pins for electrical characteristics.

By default the `PB4` and `PB5` pins are configured for the JTAG debugger. These can be used for general PIO after setting an internal bit. See disabling JTAG pins.

The logic levels are set by the voltage on the VDDIO pin (usually 3.3 V). `PA12`–`PA14` and `PA26`–`PA31` can sink/source 4 mA. The USB pins (`PB10`–`PB11`) can sink/source 30 mA. The rest can only sink/source 2 mA of current.

The pullup and pulldown resistors are typically 100 kΩ. On reset, pullup resistors are enabled.

2.2.3 USART/UART

The SAM4S has two USARTs and two UARTs. The USARTs can emulate a UART, have hardware flow control, and have a better driver so they are recommended if you need a UART interface.

2.2.4 PWM

The SAM4S can generate four independent PWM signals. There are restrictions on which SAM4S pins they come out on. Note, the PWMLx and PWMHx signals are complementary (i.e., one is low while the other is high).

2.2.5 TWI

The SAM4S has two TWI peripherals (that can act as a master and a slave) with dedicated TWD and TWCK pins. External pull-up resistors are required. TWI1 shares pins with JTAG; you will need to disable JTAG in software.

2.2.6 SPI

The SAM4S has a single SPI peripheral with dedicated SCK, MISO, and MOSI pins. Any PIO pin can be used for the chip select[1].

[1]For high speed operation (not needed for this assignment), you should use one of the dedicated chip select pins.

2.2.7 ADC

The SAM4S has a single ADC with a multiplexer to select one of a number of analogue inputs. It can sample at 1 MHz.

2.2.8 USB

The SAM4S has a single USB peripheral connected to the DDP and DDM pins. 27 ohm series termination resistors are required, placed close to the SAM4S.

2.3 Other chips

2.3.1 DRV8833 dual H-bridge

The DRV8833's datasheet is available at https://www.ti.com/product/DRV8833.

The H-bridge has four modes: forward, reverse, slow decay (brake), and fast decay (coast). With slow decay mode, the motor is shorted so that it stops faster. With fast decay mode, the motor is open-circuited and so it takes longer to stop. However, there is little difference in practice, due to friction in the gearbox.

If you want control over fast decay and slow decay in both forward and reverse you will need **four** independent PWM signals. The SAM4S can provide four independent PWM signals but be **careful** since PWMxH and PWMxL are complementary signals driven from the same PWM source.

If you are clever, you can drive the H-bridge with only two PWM channels. If you are not so clever, you will have fast decay in one direction and slow decay in the other.

The capacitor connected to the bootstrap pin must be rated for 16 V. The datasheet recommends an X7R dielectric.

2.3.2 NRF24L01+ radio

The nRF24 module we provide is actually a tiny PCB with all of the high frequency analogue components populated for you. This module breaks out the SPI communication pins, the power supply pins, and two signal pins that you will need to connect to your microcontroller. The CE and IRQ pins can both go to general PIO pins while the SPI pins (MOSI, MISO, SCLK, CSN) need to be connected to the SAM4S SPI peripheral. The nRF24L01 datasheet and other documentation for the breakout board is available at https://www.sparkfun.com/products/691.

As this radio is ultimately an analogue circuit and any noise on the power supply can affect the signal quality, we recommend using a separate 3V3 regulator and using a low pass filter to provide the best power. Instead of a resistor, a ferrite bead is better.

You cannot have a PCB plane near the antenna of the radio otherwise the **range will be severely limited**.

The radio operates around 2.4 GHz and has 128 programmable channels, each of 1 MHz. Note, some of these channels use the same spectrum as Bluetooth and WiFi. A 5 byte address is appended to the start of each transmission and the receiver will only respond when the address matches.

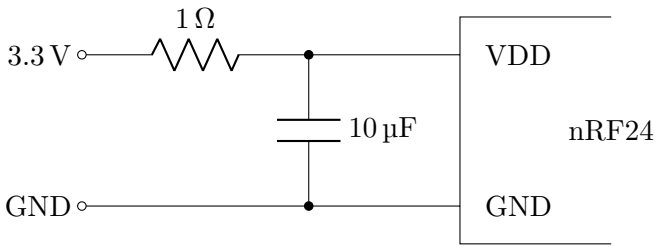


Figure 2.1: Power supply filtering; a ferrite bead is better than a resistor.

The radio interfaces to the SAM4S using the SPI bus. The IRQ pin is driven low to indicate a packet has been received.

2.3.3 MPU9250 IMU

This contains a three axis accelerometer, a three axis gyroscope, and a three axis magnetometer. It appears that the magnetometer has been bolted on to the accelerometers/gyroscopes and requires more hoop jumping in software to make it work. Documentation is available at https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/.

It has two different I2C addresses (0x68 and Ox69) depending on the state of the AD0 pin.

2.3.4 Buck converter

The buck converter is a switch-mode regulator that converts the battery voltages to 5 V DC.

2.3.5 Voltage regulators

There are many flavours of `voltage regulator`. Some are better for digital applications, some are better for analogue applications, some are better for low power applications, etc.

If you are using a voltage regulator with an enable pin, do not forget to allow for the time for the output voltage to ramp up. This can be tens of milliseconds depending on the capacitive load and current draw.

Note, some regulators have pins that you must not connect. Some have multiple pins for the same purpose; these must all be connected.

2.4 External components

2.4.1 Battery

The Wacky Racer batteries are Turnigy 5-cell 6 V, 2300 mAh, NiMH with a three pin JST connector. To preserve the battery life it is imperative to not draw current when the battery voltage is below 5 V. Note, when fully charged, the battery voltage may be 7.5 V.

The battery uses a standard RC servo connector: 3 pin 0.1” (pin 1 GND, pin 2 5V, pin 3 NC). We suggest connecting both the first and third pins to ground to allow the battery to be plugged in in

both orientations. The 461 Altium library has a component named 'Battery_HEADER_3pin' that is suitable to use.

2.4.2 LED Tape

The LED tape is a daisy-chained string of WS2812 smart LEDs. These LEDs use a clever mechanism to pass the RGB data down the string[2] with different pulse lengths for 1 and 0. They are connected as shown in Figure 2.2.

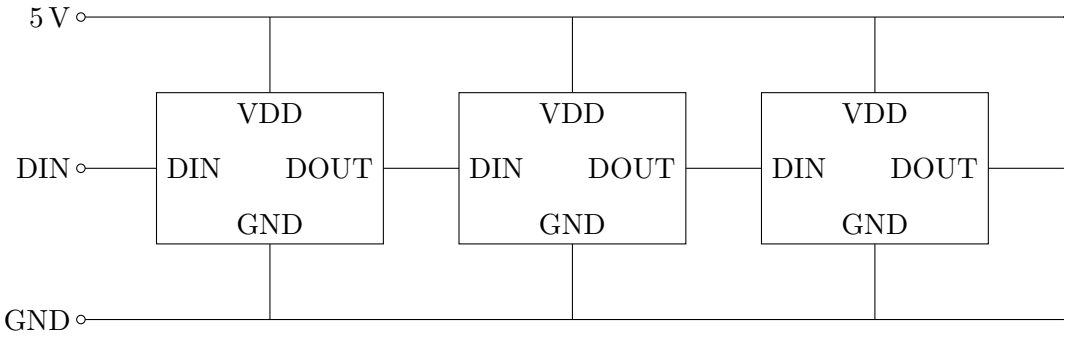
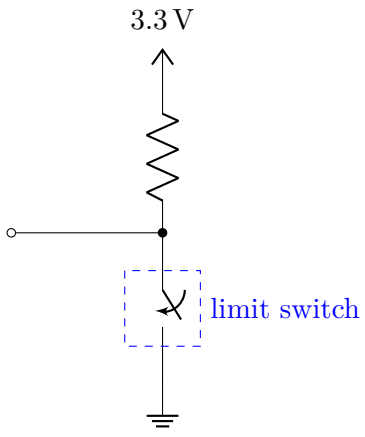


Figure 2.2: Chaining of smart LEDs in the LED-tape.

This means you will need to provide a three pin header (0.1" standard header) with the pinout: pin 1 5 V, pin 2 signal, pin 3 ground. Each individual LED has a maximum supply current of 60 mA (20 mA per red, green, blue channel). We will provide up to half a metre of LED tape to each car or hat, giving a maximum number of 30 LEDs to be driven.

2.4.3 Bumper

The provided bumper senses the contact through a simple limit switch. This switch is normally open and on contact will close.



Note that the pullup resistor shown could simply be the internal pullup resistor on a PIO pin. The connector for the limit switch should be a simple 2 pin 0.1" header.

[2]Each element consists of red, green, and blue LEDs with an 8-bit shift register to control the brightness for each colour. These shift registers are chained together.

2.4.4 Buzzer

The buzzers supplied are passive piezo-electric devices. Applying a voltage across the two terminals causes the element to deform. Applying an alternating current to the device generates an audible tone. The larger the applied voltage differential, the louder the tone becomes. There are plenty of example circuits for piezo buzzers available online. Note, you need to be able to charge and discharge the capacitance of the pizeocrystal.

2.4.5 Motors

The motors available in the provided chassis are 6 V DC motors. These have a low DC resistance and will take as much current as the DRV8833 motor driver can supply. You need to add connectors for the motors, either 2 pin 0.1" headers or screw terminals are suggested.

2.4.6 Connectors

1. USB micro or mini connector for debugging
2. 3 pin 0.1" for LED tape (pin 1 5 V, pin 2 signal, pin 3 ground)
3. 2 pin 0.1" for bumper (pin 1 switch, pin 2 ground)
4. 10 pin IDE for serial wire debug
5. 3 pin JR battery connector (pin 1 GND, pin 2 VBAT, pin 3 GND)
6. motor connectors (for Racer)
7. connectors for dastardly stuff

2.5 Other components

1. We recommend that you use components in the ECE Altium library. These are stocked in the SMT lab. For any other components you may require, see Scott Lloyd in the SMT lab.

2.6 Reading datasheets

The manufacturer will tell you what they are proud of on the first page. They will omit details of poor aspects.

There is a knack in reading a large datasheet. Here's what I do first:

1. Look at the example circuits.
2. Read the absolute maximum ratings.
3. Read the pin definitions.

Chapter 3

Schematic

3.1 Altium tutorial

See the ENCE461 Learn page for the Altium tutorial.

3.2 Suggestions

1. If you get the schematic wrong, the PCB will be wrong, and you will have rework grief (see rework).
2. A clear schematic is important for debugging.
3. In general, you should aim to have inputs on the left, outputs on the right, higher voltages at the top.
4. Add notes to explain anything non-obvious.

3.3 Check list

You must check the schematic carefully. I use a highlighter to mark every wire and component. If you get the schematic wrong, the PCB will be wrong, and you will have to do much rework, see Figure **??**.

1. The SPI signals for the radio are connected to the correct MCU pins.
2. The PWM signals for the motor are connected to the correct MCU pins. Note, the PWMLx and PWMHx pins are complementary and cannot be driven independently.
3. All the MCU VDD pins need to be powered.
4. All the MCU GND pins need to be powered.
5. Avoid connecting to `PB4` and `PB5` (say for TWI1). If you do you will need to disable JTAG in software.

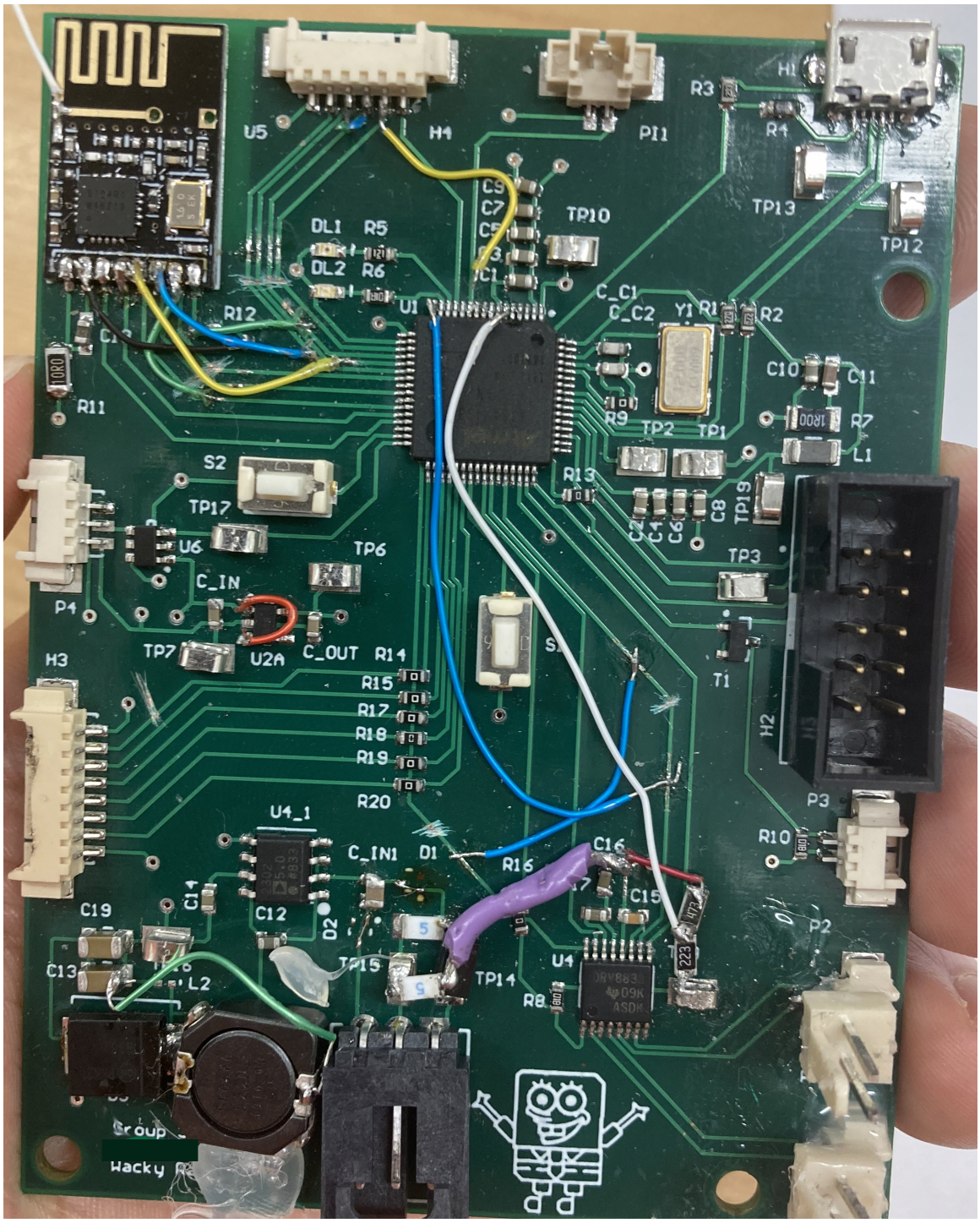


Figure 3.1: Example of a PCB where not much care was taken with the schematic resulting in a lot of rework. And yes, the board worked in the end.

Chapter 4

PCB layout

Before you lay out your PCB, an hour spent checking your schematic will avoid many hours of testing and rework grief! I annotate a printed schematic with a highlighter to ensure I have considered everything.

4.1 Altium tutorial

The Altium tutorial for creating a PCB can be found on the ENCE461 Learn page.

4.2 Placement

1. Keep small signal analogue components (radio) well away from digital electronics and power electronics.
2. Place local decoupling capacitors to minimise the loop area.
3. Place bulk capacitors close to where power comes from.
4. Keep the crystal close to the MCU.
5. Place switches so they can be pushed.
6. Place LEDs so they can be seen.
7. Place USB connector so it can be connected to.
8. Place connectors on edge of PCB with wires going away from the board.

4.3 Power supplies

1. Use planes for power distribution.
2. If cannot use power plane for power connection, make trace as wide[1] as possible.
3. Do not put splits in planes[2].

[1] To reduce inductance and resistance.
[2] Unless you know what you are doing.

4. Keep planes away from the radio antenna; otherwise the radio range is limited.
5. Before 'pouring' a polygon to create a power or ground plane connect the nets with tracks and vias.

4.4 Signal traces

1. Use microstrips for any signal above 50 kHz.
2. Keep signal traces apart to reduce crosstalk (3-W rule).
3. Avoid signal traces jumping layers (especially for signals above 50 kHz). If you do, use vias close to ends of traces.
4. Use tented[3] vias under components with metal pads.

4.5 Recommended layouts

For any switching power supply, sensitive analogue, or BGA fanout, the chip manufacturer usually provides a recommended layout.

4.6 Check list

1. Check your schematic. Then get someone else to check your schematic. Query every component.
2. Perform a DRC (Design Rule Check)].
3. Add PCB test points, test points, and more test points. You will need one for every signal and power supply. **You have been warned!** If you don't believe me, compare the size of an oscilloscope probe to a surface mount IC pin and see if you can hold the probe in place.
4. Label the test points on the silk screen with meaningful names.
5. Add ground test points that you can clip an oscilloscope ground lead to. Keep these away from other test points to avoid shorting with the oscilloscope ground clip.
6. Check mounting holes.
7. Check component footprints, especially connectors, voltage regulators, and surface mount transistors by placing the parts on a print-out of the PCB design.
8. Ensure that power supply traces are wide as possible, preferably planes, to reduce their inductance.
9. Ensure that high current traces are wide as possible to reduce their resistance.
10. Do not connect IC pins directly between the pads since it is not possible to tell if a solder blob is accidental or deliberate.
11. Insert vias to ground plane to ensure that all parts are connected. Do not leave unconnected copper.

[3] These have a layer of insulating solder resist.

12. If not using plated through vias do not place vias under components.
13. If placing vias under components with exposed metal pads, make sure the vias are "tented" (covered in solder mask) to prevent unexpected shorts.
14. Stitch power or ground planes where vias or tracks cut them up.
15. If a track is of width W it should be separated from the next track by a width 2W to reduce crosstalk. This is the 3-W rule since the track centres are 3W. An exception is for differential signals; these should be routed close together with a uniform spacing.
16. Do not run a microstrip traces right on the edge of the PCB. If track is of width W it should be at least W from the edge to reduce electric field fringing effects.
17. Use large traces for pads on connectors that may be subject to mechanical forces (power jacks, pluggable terminals, etc.) to prevent trace cracking.
18. If you have high voltages check clearances.
19. Check pad to track clearances to avoid potential solder bridges.
20. Check that it is possible to top-solder through-hole components on non-plated-through boards.
21. Ensure like signals are grouped, data lines, address lines, control lines, analogue signals, etc.
22. Check the thermal path for parts that get hot.
23. Run tracks over reference (power or ground) planes to create a microstrip trace; avoid tracks running over cuts in a reference plane.
24. Check that hole diameters are larger than the pin diameters for connectors and through hole. 25% over the lead diameters is typical. Remember to look at the diagonal dimension for square pins or your holes may be too small.
25. Check the drill file report to make sure the hole sizes are what you would expect. There should be no holes with sizes of zero, no weird sizes, no super large sizes unless required by the design.
26. Check that connections of thermal vias have clearance to other traces and pads.
27. Pay close attention to clearances on internal planes of multilayer boards; shorts on these planes can only be removed with precision drilling.
28. Number pins on connectors, big ICs, selection jumpers, etc. For connectors, number enough pins that pin ordering is obvious. For large ICs consider adding tickmarks every 5 pins.
29. Make the pad for pin 1 of every IC square. Also consider putting a silkscreen dot next to pin 1.
30. Leave at least 20–50 mils (0.5–1.3 mm) clearance between components and the edge of the board. This makes it less likely that inaccuracies in board fabrication will cause part of a pad to get chopped off.
31. Check for component clearances from enclosure including heights.
32. Look at each net individually to ensure that it doesn't take an overly long path around the board. Many layout tools have a highlight net feature that makes this a fast process.
33. Don't let the silkscreen overlap pads. Try to avoid having silkscreen overlapping via holes or areas of the board that will be routed away (large holes, slots, tabs, etc.), since legibility will be poor.
34. If ICs are being soldered with solder paste ensure that there is a solder mask between pads.

4.7 Design rule checks

This section considers common Altium design rule violations. For more information see the Altium wiki.

4.7.1 PCB manufacture violations

You do not want any of these. If a trace is too thin then etching may produce a hole in it. It two traces are too close together, they may end up joined.

4.7.2 PCB assembly violations

SMD neck-down constraint

Traces connecting to SMD pads often need to be reduced in thickness to minimise heat transfer from the SMD pad (unequal PAD temperatures can lead to component tombstoning or rotation in the oven). The traces widths should be a maximum of 50 % of the width of the pad and should be of the same size to prevent component movement due to unequal pad temperatures. The length of the neck-down can be as short as 0.5 mm.

This is not so critical for integrated circuits with many pins (except if a solder mask is not used since solder might flow away from the pad).

Silk to solder mask

Here Altium thinks the silk screen is being printed over a pad; most manufacturers detect this and delete the offending text.

Minimum solder mask sliver constraint

This occurs with chips having pins that are close together. It is a warning that the PCB manufacturer may not be able to make the thin sliver of solder mask that is applied between pads to reduce the chances of having a solder blob between the pins. For a prototype it is not a concern since the solder blob can be easily removed with a small soldering iron.

4.7.3 Signal integrity violations

SMD pad to plane constraint

What this means is that you have placed a via to a power or ground plane too far from a SMD pad. Simply add another via closer to the pad. There is no need to skimp on vias for power supply connections.

This violation indicates you that you are adding unnecessary inductance by increasing the loop area and thus compromising signal integrity.

Unfortunately, Altium does not detect when you use a power decoupling capacitor to mitigate this inductance and thus will report false-positives. For some reason, whenever there is a decoupling

capacitor pad between an IC pad and a via to a plane, it cannot calculate the distance of the trace and reports it as being about 2.3 m! So, ignore this error.

The goal is to place the decoupling capacitor to minimise the area of the loop formed by the current flowing from the capacitor to the IC power and ground pins. Multiple vias to power/ground planes should then be placed on the other side of the capacitor.

Chapter 5

PCB assembly

5.1 SMT lab induction

You cannot use the SMT lab unless you have performed a lab induction.

5.2 Solder pasting

Solder paste needs to be applied to each pad. Usually this is applied using a stencil but for small boards, it can be applied using a syringe. The solder paste must be fresh.

5.3 Component placement

The components are placed onto the pasted pads using a pick-and-place machine. The key is to get the orientation of chips, diodes, and LEDs correct.

It is easy to orient the MCU incorrectly. The big dot **does not mark pin 1**[1], see Figure 5.1.

5.4 Reflow oven

When the board has been populated, the PCB is placed in the reflow oven. Thus is preheated to 150° C so you must use oven gloves to put your board in the oven. At the end of the process, remove your PCB using oven gloves. You do not need to clean the board.

5.5 Visual inspection

Look for solder bridges, see Figure 5.2, especially between IC pins. The bridges can be removed with a fine-tipped soldering iron.

[1] I think they add this to sell more chips...

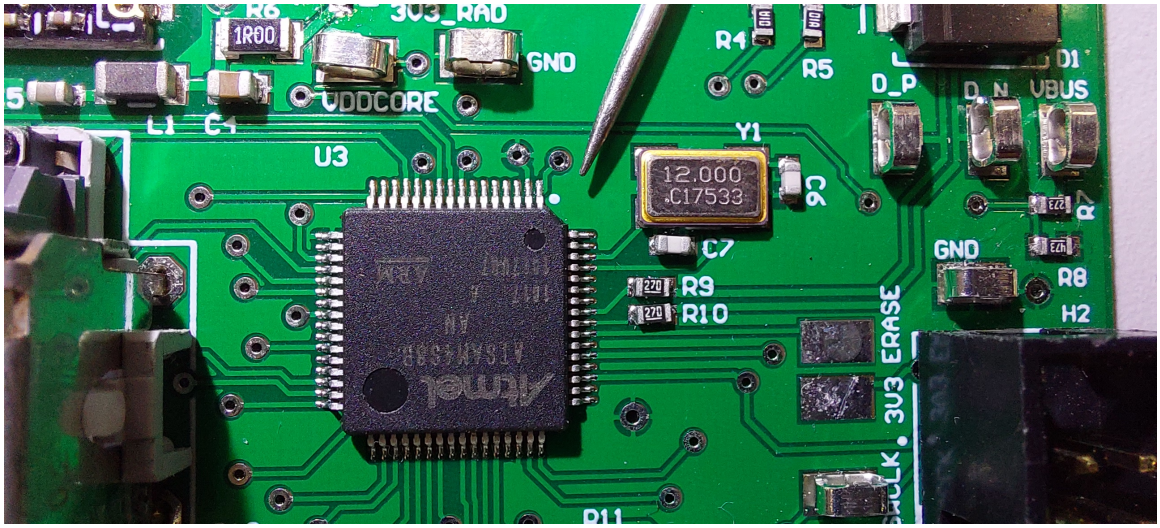


Figure 5.1: Pin 1 of the SAM4S is marked by the chamfer and the small dot.

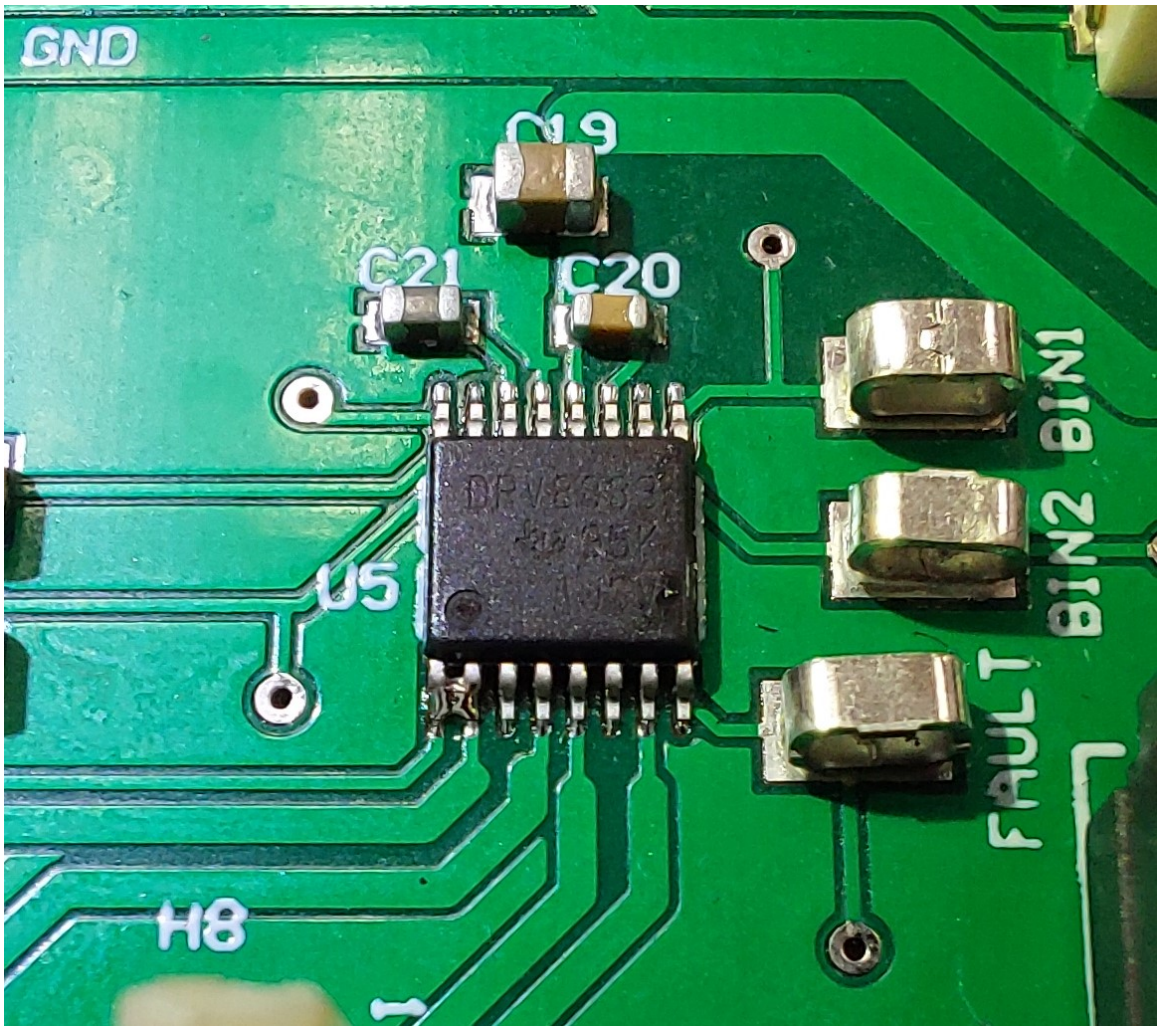


Figure 5.2: A solder bridge shorting pins 1 and 2 of the H-bridge. This can easily occur if there is too much solder paste.

Chapter 6

Software installation

Do not underestimate the effort required to flash your first LED. You require:

- A computer with `git` installed and a useful shell program such as `bash`. UC has a mirror for a variety of Linux distributions; we recommend Ubuntu or Mint.
- A cloned copy of the Wacky Racers git repository, see project cloning.
- A working ARM toolchain (arm-none-eabi-gcc/g++ version 4.9.3 or newer), see toolchain.
- An ST-link programmer and 10-wire ribbon cable for programming. You can get the adaptor from Scott Lloyd in the SMT lab. You will need to make your own cable (For a grey ribbon cable, align the red stripe with the small arrow denoting pin 1 on the connector. For a rainbow ribbon cable, connect the brown wire to pin 1.). There are two variants of the ST-link programmer with **different pinouts** so you may need to customise your programming cable.
- Plenty of gumption.

6.1 Git

Your group leader should fork the Wacky Racers project template. This creates your own group copy of the project on the eng-git server that you can modify, add members, etc.

Each group member then clones the group project.

6.1.1 Project forking

The template software is hosted on the eng-git `Git` server. To fork the template:

1. Go to https://eng-git.canterbury.ac.nz/wacky-racers/wacky-racers.
2. Click the 'Fork' button. This will create a copy of the main repository for the project.
3. Click on the 'Settings' menu then click the 'Expand' button for 'Sharing and permissions'. Change 'Project Visibility' to 'Private'.
4. Click on the 'Members' menu and add group members as Developers.

6.1.2 Project cloning

Once your project has been forked from the template project, each group member needs to clone it. This makes a local copy of your project on your computer.

If you are using an ECE computer, it is advised that you clone the project on to a removable USB flash drive. This will make git operations and compilation 100 times faster than using the networked file system.

There are two ways to clone the project. If you are impatient and do not mind having to enter a username and password for every git pull and push operation use:

```
$ git clone https://eng-git.canterbury.ac.nz/groupleader-userid/wacky-racers.git
```

Otherwise, set up `ssh-keys` and use:

```
$ git clone git@eng-git.canterbury.ac.nz:groupleader-userid/wacky-racers.git
```

You can have several different cloned copies of your project in different directories. Sometimes if you feel that the world, and `git` in particular, is against you, clone a new copy, using:

```
$ git clone https://eng-git.canterbury.ac.nz/groupleader-userid/wacky-racers.git
↪  wacky-racers-new
```

6.2 Toolchain

The toolchain comprises the compiler, linker, debugger, C-libraries, and OpenOCD.

The toolchain is installed on computers in the ESL and CAE. It should run under both Linux and Windows. If there is a problem ask the technical staff.

The toolchain can be downloaded for Windows, Linux, and macOS from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads. For Linux and macOS, however, it is better to install the toolchain using your system's package manager: see the instructions in the following subsections.

6.2.1 Toolchain for Linux

First, if using Ubuntu or Mint, ensure the latest versions are downloaded:

```
$ sudo apt update && sudo apt upgrade
```

Then, install the compiler:

```
$ sudo apt install gcc-arm-none-eabi
```

Install the C and C++ libraries:

```
$ sudo apt install libnewlib-arm-none-eabi libstdc++-arm-none-eabi-newlib
```

Install the debugger, GDB:

```
$ sudo apt install gdb-multiarch
```

Install OpenOCD:

```
$ sudo apt install openocd
```

6.2.2 Toolchain for macOS

For macOS machines that have homebrew installed, you can use the following command:

```
$ brew install openocd git
$ brew cask install gcc-arm-embedded
```

6.3 IDE

I am content with using the command-line and emacs as a text editor but I guess that you are not. If you are familiar with geany I suggest that you use that. If you want bells and whistles, I suggest using Visual Studio Code. While it is written by Microsoft, it is free and runs on Windows, Linux and macOS.

6.3.1 Visual Studio Code

VS Code is a modern and highly versatile text editor. With the right extensions, it can be used to develop code for just about anything! This project has been configured for VS Code on the ESL computers (but its relatively easy to modify it to work on a home computer, be it Windows, Linux, or Mac). For the simplest use, simply go `File -> Open Folder` and point it at the wacky-racers repository. This will give you access to the build tools (Ctrl-Shift-B) that have been setup. These must be run from inside a program (i.e., ledflash1.c). Finally, opening a program and pressing F5 will launch a debugging session that will allow the use of breakpoints and variable inspection.

When building your programs, the `BOARD` configuration variable must be set. In VS Code this is done by choosing your C++ configuration (bottom right of the window) which by default will be either hat or racer.

As a side note, the compilation and debugging requires the installation of two VS Code extensions:

1. C/C++ (Microsoft)
2. Native Debug (Web Freak)

VS Code should automatically prompt you to install these when you open the wacky-racers directory.

6.4 Using Windows in the ESL

On the Windows machines in the ESL, the toolchain is located at `C:\ence461\tool-chain`. This needs to be added to your session's `PATH` variable so toolchain commands can be run without needing to specify their full path. See `ARM-GCC in the DSL` on ECEWiki for more information.

You can use Windows Command Prompt similarly to a Linux Shell. To open, press `Windows Key` + `R`, type `cmd`, and `Enter`. Then run `C:\ence461\tool-chain` inside the CMD window to source the toolchain. Although this guide is written for the Linux shell, the commands will also work in Command Prompt provided you observe the following differences:

- To change between drives (e.g. `C:` and `D:`), pass the `/d` switch to the `cd` command.

- Instead of `export VARIABLE=value`, use `set VARIABLE=value`. Also you cannot set an environment variable when you run a command, so `BOARD=hat make` won't work; run `set BOARD=hat` and `make` in separate commands instead.
- To run GDB, you must invoke the complete command name `arm-none-eabi-gdb`.
- In path names, `/` is replaced by \ (backslash).

If you are using Windows on the ESL machines and want to use VS Code, you can run `vscode.bat` located in the top level of the `wacky-racers` repository. This will open VS Code with the repository added to the workspace and the necessary toolchain added to `PATH`.

Chapter 7

First program

Your first program to test your board should only flash an LED (the hello world equivalent for embedded systems). The key to testing new hardware is to have many programs that only do one simple task each.

Before you run your first program, you need to:

1. Install the toolchain.
2. Clone the Wacky Racers git repository, see project cloning.
3. Set up the PIO definitions for your board, see configuration.
4. Compile your program, see compilation.
5. See if your SAM4S is running, see OpenOCD.
6. Configure the SAM4s to boot from flash memory, see booting from flash memory.
7. Program the SAM4s, see programming.

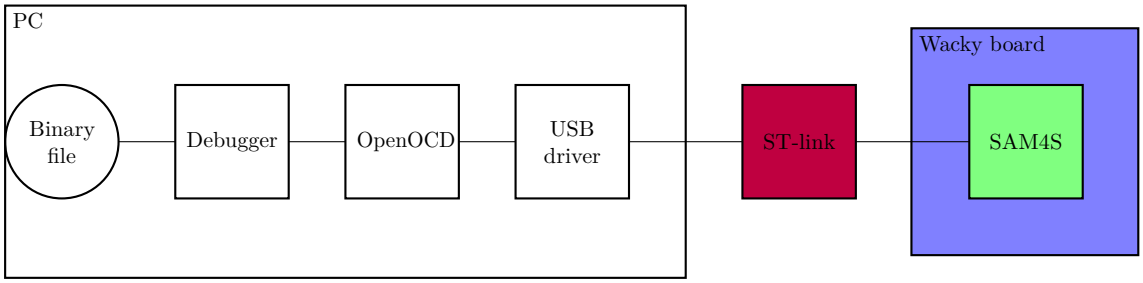


Figure 7.1: How OpenOCD interacts with the debugger and the SAM4S.

7.1 Connecting with OpenOCD

`OpenOCD` is used to program the SAM4S, see Figure 7.1.

For this assignment, we use a ST-link programmer to connect to the SAM4S using serial wire debug (SWD). This connects to your board with a 10-wire ribbon cable and an IDC connector.

1. Before you start, disconnect the battery and other cables from your PCB.

2. Connect a 10-wire ribbon cable from the ST-link programmer to the programming header on your PCB. This will provide 3.3 V to your board so your green power LED should light.

3. Open a **new terminal window, e.g., bash** and start `OpenOCD`.

```
$ cd wacky-racers
$ openocd -f src/mat91lib/sam4s/scripts/sam4s_stlink.cfg
```

All going well, the last line output from `OpenOCD` should be:

```
Info : sam4.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Congrats if you get this! It means you have correctly soldered your SAM4S. If not, do not despair and do not remove your SAM4S. Instead, see troubleshooting.

7.2 LED flash program

For your first program, use `wacky-racers/src/test-apps/ledflash1/ledflash1.c`. The macros `LED1_PIO` and `LED2_PIO` need to be defined in `target.h` (see configuration).

```
/* File:   ledflash1.c
   Author: M. P. Hayes, UCECE
   Date:   15 May 2007
   Descr:  Flash an LED
*/
#include <pio.h>
#include "target.h"
#include "pacer.h"

/* Define LED flash rate in Hz.  */
#define LED_FLASH_RATE 2

/*
    This test app is the faithful blinky program.  It works as follows:
    1. set the LED pin as an output low (turns on the LED if LED active low).
    2. initialize a pacer for 200 Hz.
    3. wait for the next pacer tick.
    4. toggle the LED.
    5. go to step 3.

    Suggestions:
    * Add more LEDs.
    * Blink interesting patterns (S-O-S for example).
    * Make two LEDs blink at two separate frequencies.
*/

int
main (void)
{
    /* Configure STATUS LED PIO as output.  */
    pio_config_set (LED_STATUS_PIO, PIO_OUTPUT_LOW);
```

```
    pacer_init (LED_FLASH_RATE * 2);

    while (1)
    {
        /* Wait until next clock tick.  */
        pacer_wait ();

        /* Toggle LED.  */
        pio_output_toggle (LED_STATUS_PIO);
    }
}
```

The LED should flash at 2 Hz. If it does not, try the PWM test program and check the frequency with an oscilloscope.

7.3 Configuration

Each board has different PIO definitions and requires its own configuration information. The wacky-racers/src/boards directory contains a configuration for the hat and one for the racer board. **You must edit these to customise your board.** Each configuration directory contains three files:

- `board.mk` is a makefile fragment that specifies the MCU model, optimisation level, etc.
- `target.h` is a C header file that defines the PIO pins and clock speeds.
- `config.h` is a C header file that wraps target.h. Its purpose is for porting to different compilers.

You will need to edit the target.h file for your board and set the definitions appropriate for your hardware. Here's an excerpt from `target.h` for a hat board:

```
/* USB  */
#define USB_VBUS_PIO PA5_PIO

/* ADC  */
#define ADC_BATTERY PB3_PIO
#define ADC_JOYSTICK_X PB2_PIO
#define ADC_JOYSTICK_Y PB1_PIO

/* IMU  */
#define IMU_INT_PIO PA0_PIO

/* LEDs  */
#define LED1_PIO PA20_PIO
#define LED2_PIO PA23_PIO
```

7.4 Compilation

Due to the many files required, compilation is performed using makefiles.

The demo test programs are generic and you need to specify which board you are compiling them for. The board configuration file can be chosen dynamically by defining the environment variable `BOARD`. For example:

```
$ cd src/test-apps/ledflash1
$ BOARD=racer make
```

If all goes well, you should see at the end:

```
   text    data     bss     dec     hex filename
  11348    2416     176   13940    3674 ledflash1.bin
```

To avoid having to specify the environment variable `BOARD`, you can define it for the rest of your session using:

```
$ export BOARD=racer
```

and then just use:

```
$ make
```

7.5 Booting from flash memory

By default the SAM4S runs a bootloader program stored in ROM. The SAM4S needs to be configured to run your application from flash memory.

If `OpenOCD` is running you can do this with:

```
$ make bootflash
```

Unless you force a complete erasure of the SAM4S flash memory by connecting the `ERASE` pin to 3.3 V, you will not need to repeat this command.

7.6 Programming

If `OpenOCD` is running you can store your program in the flash memory of the SAM4S using:

```
$ make program
```

When this finishes, one of your LEDs should flash. If so, congrats! If not, see troubleshooting.

To reset your SAM4S, you can use:

```
$ make reset
```

7.7 Makefiles

Compilation is performed using makefiles, since each application requires many files. Rather than having large makefiles, a heirarchy of makefile fragments is employed. Dependencies are automatically generated (so not all the files have to be recompiled each time).

A board description can be selected with the `BOARD` environment variable. This can be defined for each command, for example,

```
$ BOARD=racer make program
```

Alternatively, this can be defined for a session:

```
$ export BOARD=racer
$ make program
```

Each of the makefiles has the following phony targets:

all — compile the application

program — compile the application and download to the MCU (you need Openocd running)

debug — start the debugger GDB (you need Openocd running)

reset — reset the MCU (you need Openocd running)

clean — delete the executable, object files, and dependency files.

Chapter 8

Test programs

There are a number of test programs in the directory wacky-racers/src/test-apps. Where possible these are written to be independent of the target board using configuration files (see configuration).

8.1 USB

To help debug your programs, it is useful to be able to print values. This can be achieved by redirecting stdio using the USB CDC protocol. For example, here's an example program wacky-racers/src/test-apps/usb_serial_test1/usb_serial_test1.c.

```
#include <stdio.h>
#include "usb_serial.h"
#include "pio.h"
#include "delay.h"

#define HELLO_DELAY_MS 500

int main (void)
{
    int i = 0;

    pio_config_set (LED_ERROR_PIO, PIO_OUTPUT_LOW);

    // Redirect stdio to USB serial
    usb_serial_stdio_init ();

    while (1)
    {
        delay_ms (HELLO_DELAY_MS);

        printf ("Hello world %d\n", i++);

        pio_output_toggle(LED_ERROR_PIO);
    }
}
```

To get this program to work you need to compile it and program the SAM4S using:

```
$ cd wacky-racers/src/test-apps/usb_serial_test1
$ make program
```

You then need to connect your computer to the USB connector on your PCB. If you are running Linux, run:

```
$ dmesg
```

This should say something like:

```
Apr 30 11:03:50 thing4 kernel: [52704.481352] usb 2-3.3: New USB device found, idVendor=03eb, idPr
Apr 30 11:03:50 thing4 kernel: [52704.481357] usb 2-3.3: New USB device strings: Mfr=1, Product=2,
Apr 30 11:03:50 thing4 kernel: [52704.482060] cdc_acm 2-3.3:1.0: ttyACM0: USB ACM device
```

Congrats if you see `ttyACM0: USB ACM device`! If not, see USB debugging.

You can now run a serial terminal program. For example, on Linux:

```
$ gtkterm -p /dev/ttyACM0
```

All going well, this will repeatedly print 'Hello world'.

If run Linux and get an error 'device is busy', it is likely that the ModemManager program has automatically connected to your device on the sly. This program should be disabled on the ECE computers. For more about this and using other operating systems, see USB CDC.

8.2 PWM

The program wacky-racers/src/test-apps/pwm_test2/pwm_test2.c provides an example of driving PWM signals.

```
/* File:   pwm_test2.c
   Author: M. P. Hayes, UCECE
   Date:   15 April 2013
   Descr:  This example starts two channels simultaneously; one inverted
           with respect to the other.
*/
#include "pwm.h"
#include "pio.h"


#define PWM1_PIO PA1_PIO
#define PWM2_PIO PA2_PIO

#define PWM_FREQ_HZ 100e3


static const pwm_cfg_t pwm1_cfg =
{
    .pio = PWM1_PIO,
    .period = PWM_PERIOD_DIVISOR (PWM_FREQ_HZ),
    .duty = PWM_DUTY_DIVISOR (PWM_FREQ_HZ, 50),
```

```
    .align = PWM_ALIGN_LEFT,
    .polarity = PWM_POLARITY_LOW,
    .stop_state = PIO_OUTPUT_LOW
};

static const pwm_cfg_t pwm2_cfg =
{
    .pio = PWM2_PIO,
    .period = PWM_PERIOD_DIVISOR (PWM_FREQ_HZ),
    .duty = PWM_DUTY_DIVISOR (PWM_FREQ_HZ, 50),
    .align = PWM_ALIGN_LEFT,
    .polarity = PWM_POLARITY_HIGH,
    .stop_state = PIO_OUTPUT_LOW
};


int
main (void)
{
    pwm_t pwm1;
    pwm_t pwm2;

    pwm1 = pwm_init (&pwm1_cfg);
    pwm2 = pwm_init (&pwm2_cfg);

    pwm_channels_start (pwm_channel_mask (pwm1) | pwm_channel_mask (pwm2));

    while (1)
        continue;

    return 0;
}
```

Notes:

1. This is for a different H-bridge module that requires two PWM signals and forward/reverse signals. You will need to generate four PWM signals or be clever with two PWM signals.
2. The `pwm_cfg_t` structure configures the frequency, duty cycle and alignment of the output PWM.
3. The frequency is likely to be too high for your motor.
4. `pwm_channels_start` is used to start the PWM channels simultaneously.

The most likely problem is that you have not used a PIO pin that can be driven as a PWM output. The SAM4S can generate four independent hardware PWM signals. See wacky-racers/src/mat91lib/pwm/pwm.c for a list of supported PIO pins. Note, `PA16`, `PA30`, and `PB13` are different options for PWM2.

To drive the motors you will need to use a bench power supply. Start with the current limit set at 100 mA maximum in case there are any board shorts. When all is well, you can increase the current limit; you will need at least 1 A.

8.3 IMU

The MPU9250 IMU connects to the SAM4S using the I2C bus (aka TWI bus). The program wacky-racers/src/test-apps/imu_test1/imu_test1.c provides an example of using the MPU9250 IMU. All going well, this prints three 16-bit acceleration values per line to USB CDC. Tip your board over, and the the third (z-axis) value should go negative since this measures the effect of gravity on a little mass inside the IMU pulling on a spring.

If you get 'ERROR: can't find MPU9250!', the main reasons are:

1. You have specified the incorrect address. Use 0x68 for `MPU_ADDRESS` in `target.h` if the AD0 pin is connected to ground otherwise use 0x69.

2. You are using TWI1. The `PB4` and `PB5` pins used by TWI1 default to JTAG pins. See disabling JTAG pins.

See also IMU checking.

Other problems:

1. If you reset the SAM4S in the middle of a transaction with the IMU, the IMU gets confused and holds the `TWD/SDA` line low. This requires recycling of the power or sending out some dummy clocks on the `TWCK/SCL` signal.

8.4 Radio

The program wacky-racers/src/test-apps/radio_tx_test1/radio_tx_test1.c provides an example of using the radio as a transmitter.

The companion program wacky-racers/src/test-apps/radio_rx_test1/radio_rx_test1.c provides an example of using the radio as a receiver.

```
/* File:   radio_rx_test1.c
   Author: M. P. Hayes, UCECE
   Date:   24 Feb 2018
*/
#include "nrf24.h"
#include "usb_serial.h"
#include "pio.h"
#include "delay.h"

#define RADIO_CHANNEL 4
#define RADIO_ADDRESS 0x0123456789LL

static void panic (void)
{
    while(1)
    {
        pio_output_toggle (LED_ERROR_PIO);
        delay_ms (400);
    }
}
```

```
int main(void)
{
    nrf24_cfg_t nrf24_cfg =
        {
            .channel = RADIO_CHANNEL,
            .address = RADIO_ADDRESS,
            .payload_size = 32,
            .ce_pio = RADIO_CE_PIO,
            .irq_pio = RADIO_IRQ_PIO,
            .spi =
            {
                .channel = 0,
                .clock_speed_kHz = 1000,
                .cs = RADIO_CS_PIO,
                .mode = SPI_MODE_0,
                .cs_mode = SPI_CS_MODE_FRAME,
                .bits = 8
            }
        };
    nrf24_t *nrf;

    // Configure LED PIO as output.
    pio_config_set (LED_ERROR_PIO, PIO_OUTPUT_LOW);
    pio_config_set (LED_STATUS_PIO, PIO_OUTPUT_HIGH);

    // Redirect stdio to USB serial.
    usb_serial_stdio_init ();

#ifdef RADIO_POWER_ENABLE_PIO
    // Enable radio regulator if present.
    pio_config_set (RADIO_POWER_ENABLE_PIO, PIO_OUTPUT_HIGH);
    delay_ms (10);
#endif

    nrf = nrf24_init (&nrf24_cfg);
    if (! nrf)
        panic ();

    while(1)
    {
        char buffer[RADIO_PAYLOAD_SIZE + 1];
        uint8_t bytes;

        bytes = nrf24_read (nrf, buffer, RADIO_PAYLOAD_SIZE);
        if (bytes != 0)
        {
            buffer[bytes] = 0;
            printf ("%s\n", buffer);
            pio_output_toggle (LED_STATUS_PIO);
```

```
        }
    }
}
```

Notes:

1. Both programs must use the same RF channel and the same address.
2. Some RF channels are better than others since some overlap with WiFi and Bluetooth.
3. The address is used to distinguish devices operating on the same channel. Note, the transmitter expects an acknowledge from a receiver on the same address and channel.
4. The number of bytes that can be transmitted is set by the `payload\size` field in the configuration structure. This has a maximum value of 32.
5. The radio 'write' method blocks waiting for an auto-acknowledgement from the receiver device. This acknowledgement is performed in hardware. If no acknowledgement is received, it retries for up to 15 times. The auto-acknowledgement and number of retries can be configured in software.
6. If the program hangs in the `panic` loop, there is no response from the radio module, check SPI connections and see SPI debugging.
7. The radio has two modes: transmit and receive. Calling `nrf24_write` switches to transmit mode and calling `nrf24_read` switches to receive mode.

If you cannot communicate between your hat and racer boards, try communicating with the radio test modules Scott Lloyd has in the SMT lab.

8.5 ADC

wacky-racers/src/test-apps/adc_usb_serial_test2/adc_usb_serial_test2.c shows how to read from two multiplexed ADC channels, specifically the hat joystick channels (although it can be adapted for different purposes). For more details see wacky-racers/src/mat91lib/adc/adc.h.

```
/* File:   adc_usb_serial_test2.c
   Author: M. P. Hayes, UCECE
   Date:   3 May 2021
   Descr:  This reads from joystick ADC channels.
           It triggers ADC conversions as each sample is read.
*/
#include <stdio.h>
#include "usb_serial.h"
#include "adc.h"
#include "pacer.h"

#define PACER_RATE 2

static const adc_cfg_t adc_cfg =
{
    .bits = 12,
    .channels = BIT (JOYSTICK_X_ADC_CHANNEL) | BIT (JOYSTICK_Y_ADC_CHANNEL),
    .trigger = ADC_TRIGGER_SW,
```

```
    .clock_speed_kHz = 1000
};


int main (void)
{
    usb_cdc_t usb_cdc;
    adc_t adc;
    int count = 0;

    // Redirect stdio to USB serial
    usb_serial_stdio_init ();

    adc = adc_init (&adc_cfg);

    pacer_init (PACER_RATE);
    while (1)
    {
        uint16_t data[2];

        pacer_wait ();

        // The lowest numbered channel is read first.
        adc_read (adc, data, sizeof (data));

        printf ("%3d: %d, %d\n", count, data[0], data[1]);
    }
}
```

The ADC can be also set up to stream data continuously but you need to use interrrupts or DMA.

8.6 Pushbutton

wacky-racers/src/test-apps/button_test1/button_test1.c shows the use of a simple button driver to read a pushbutton. This driver does not debounce the button.

```
/* File:   button_test1.c
   Author: M. P. Hayes, UCECE
   Date:   18 Dec 2021
   Descr:  Simple button test demo without debouncing
*/
#include "mcu.h"
#include "pio.h"
#include "pacer.h"

#define PACER_RATE 100

int
main (void)
{
```

```
    /* Configure LED PIO as output.  */
    pio_config_set (LED_STATUS_PIO, PIO_OUTPUT_LOW);

    /* Configure button PIO as input with pullup.  */
    pio_config_set (BUTTON_PIO, PIO_INPUT_PULLUP);

    pacer_init (PACER_RATE);

    while (1)
    {
        /* Wait until next clock tick.  */
        pacer_wait ();

        if (pio_input_get (BUTTON_PIO))
            pio_output_high (LED_STATUS_PIO);
        else
            pio_output_low (LED_STATUS_PIO);
    }
}
```

wacky-racers/src/test-apps/button_test2/button_test2.c shows the use of a simple button driver to read a pushbutton. This driver does button debouncing and state-transition detection. For more details see wacky-racers/src/mmculib/button/button.h.

```
/* File:   button_test2.c
   Author: M. P. Hayes, UCECE
   Date:   3 May 2021
   Descr:  Simple button test demo using debouncing
*/
#include "mcu.h"
#include "led.h"
#include "pio.h"
#include "pacer.h"
#include "button.h"

#define BUTTON_POLL_RATE 100

/* Define LED configuration.  */
static const led_cfg_t led1_cfg =
{
    .pio = LED_STATUS_PIO,
    .active = 1
};


/* Define button configuration.  */
static const button_cfg_t button1_cfg =
{
    .pio = BUTTON_PIO
};
```

```
int main (void)
{
    led_t led1;
    button_t button1;

    /* Initialise LED.  */
    led1 = led_init (&led1_cfg);

    /* Turn on LED.  */
    led_set (led1, 1);

    /* Initialise button.  */
    button1 = button_init (&button1_cfg);

    button_poll_count_set (BUTTON_POLL_COUNT (BUTTON_POLL_RATE));

    pacer_init (BUTTON_POLL_RATE);

    while (1)
    {
        pacer_wait ();

        button_poll (button1);

        if (button_pushed_p (button1))
        {
            led_toggle (led1);
        }
    }
    return 0;
}
```

8.7 LEDtape

wacky-racers/src/test-apps/ledtape_test1/ledtape_test1.c shows how to drive the LEDtape.

```
/* File:   ledtape_test11.c
   Author: M. P. Hayes, UCECE
   Date:   30 Jan 2020
   Descr:  Test ledtape
*/

#include <pio.h>
#include "target.h"
#include "pacer.h"
#include "ledtape.h"

#define NUM_LEDS 20
```

```
/*
    This test app shows how to program the WS2318B LED tape.
    In this example the ledtape_write() function is used to send the appropriate
    GRB color information down the LED tape. It takes care of all the tricky
    timings, you just need to feed it an array of color values.

    ledtape_write(
        pin - This is the pin you want to send the data on. Should be connected
              to your level shifter.
        data - This is the array of GRB data (8 bit values, 24 bits per pixel).
        size - This is the size of the array in bytes.
    )

    You will also see use of the pacer module. This isn't really needed, but
    makes it easier to see the data packets being sent if you put the data
    signal on an oscilloscope.

    Suggestions:
    * Change the number of LEDs to match yours (or move the NUM_LEDS definition
      to your target file)
    * Change the color being set on all the LEDs.
    * Make an interesting pattern.
    * Make the pattern scroll down the LEDs (see ledtape_test2 for an option).
*/


int
main (void)
{
    uint8_t leds[NUM_LEDS * 3];
    int i;

    for (i = 0; i < NUM_LEDS; i++)
    {
        // Set full green  GRB order
        leds[i * 3] = 255;
        leds[i * 3 + 1] = 0;
        leds[i * 3 + 2] = 0;
    }

    pacer_init(10);

    while (1)
    {
        pacer_wait();

        ledtape_write (LEDTAPE_PIO, leds, NUM_LEDS * 3);
    }
}
```

wacky-racers/src/test-apps/ledtape_test2/ledtape_test2.c uses a more sophisticated means of controlling the LEDs.

```
/* File:   ledtape_test2.c
   Author: B Mitchell, UCECE
   Date:   14 April 2021
   Descr:  Test ledtape
*/

#include <pio.h>
#include "target.h"
#include "pacer.h"
#include "ledbuffer.h"

#define NUM_LEDS 15

/*
    This is an alternative method for driving the LED tape using the ledbuffer
    module that is included in the ledtape driver.

    The buffer acts like a small framebuffer upon which you can set RGB values
    at specific positions (not GRB, it handles the translation automatically).
    It also makes it easy to make patterns, shuffle them allow the strip, and
    clear it later. See ledbuffer.h for more details (CTRL-Click it in VS Code).
*/

int
main (void)
{
    bool blue = false;
    int count = 0;

    ledbuffer_t* leds = ledbuffer_init(LEDTAPE_PIO, NUM_LEDS);

    pacer_init(30);

    while (1)
    {
        pacer_wait();

        if (count++ == NUM_LEDS)
        {
            // wait for a revolution
            ledbuffer_clear(leds);
            if (blue)
            {
                ledbuffer_set(leds, 0, 0, 0, 255);
                ledbuffer_set(leds, NUM_LEDS / 2, 0, 0, 255);
            }
            else
```

```
            {
                ledbuffer_set(leds, 0, 255, 0, 0);
                ledbuffer_set(leds, NUM_LEDS / 2, 255, 0, 0);
            }
            blue = !blue;
            count = 0;
        }

        ledbuffer_write (leds);
        ledbuffer_advance (leds, 1);
    }
}
```

Chapter 9

Application

Here are my recommendations for your Hat or Racer application:

1. Build and test your programs incrementally.
2. Write separate test programs that you can run when your application fails to work. When debugging embedded systems, it is much easier to use a small test program rather than a large application.
3. It is easier to use a paced loop and poll the peripherals than using interrupts.
4. Disable the watchdog timer until you have robust code.

Chapter 10

Libraries

`mmculib` is a library of C drivers, mostly for performing high-level I/O. It is written to be microcontroller neutral.

`mat91lib` is a library of C drivers specifically for interfacing with the peripherals of Atmel AT91 microcontrollers such as the Atmel SAM4S. It provides the hardware abstraction layer.

If you find a bug or would like additional functionality let us know.

10.1 PIO pins

mat91lib provides efficient PIO abstraction routines in wacky-racers/src/mat91lib/sam4s/pio.h. Each pin can be configured as follows:

```
    PIO_INPUT,              /* Configure as input pin.  */
    PIO_PULLUP,             /* Configure as input pin with pullup.  */
    PIO_PULLDOWN,           /* Configure as input pin with pulldown.  */
    PIO_OUTPUT_LOW,         /* Configure as output, initially low.  */
    PIO_OUTPUT_HIGH,        /* Configure as output, initially high.  */
    PIO_PERIPH_A,           /* Configure as peripheral A.  */
    PIO_PERIPH_A_PULLUP,    /* Configure as peripheral A with pullup.  */
    PIO_PERIPH_B,           /* Configure as peripheral B.  */
    PIO_PERIPH_B_PULLUP,    /* Configure as peripheral B with pullup.  */
    PIO_PERIPH_C,           /* Configure as peripheral C.  */
    PIO_PERIPH_C_PULLUP     /* Configure as peripheral C with pullup.  */
```

Here's an example:

```
#include "pio.h"

// Configure PA0 as an output and set default state to low.
pio_config_set (PIO_PA0, PIO_OUTPUT_LOW);

// Set PA0 high.
pio_output_high (PIO_PA0);

// Set PA0 low.
```

```
  pio_output_low (PIO_PA0);

  // Set PA0 to value.
  pio_output_set (PIO_PA0, value);

  // Toggle PA0.
  pio_output_toggle (PIO_PA0);

  // Reconfigure PA0 as an output connected to peripheral A.
  pio_config_set (PIO_PA0, PIO_PERIPH_A);

  // Reconfigure PA0 as an input with pullup enabled.
  pio_config_set (PIO_PA0, PIO_INPUT_PULLUP);

  // Read state of PIO pin.
  result = pio_input_get (PIO_PA0);
```

Note, you can reconfigure a PIO pin on the fly. For example, you may want the pin to be driven by the PWM peripheral and then at some stage forced low. To do this, use `pwm\_config\_set`.

10.2 Delaying

mat91lib provides a macro `DELAY_US` in wacky-racers/src/mat91lib/delay.h for a busy-wait delay in microseconds (this can be a floating point value). The CPU busy-waits for a precomputed number of clock cycles. The argument should be a constant so the compiler can compute the number of clock cycles.

mat91lib also provides a function `delay_ms` for a busy-wait delay in milliseconds (this must be an integer). All this function does is call `DELAY_US (1000)` the required number of times.

An example program is wacky-racers/src/test-apps/delay_test1/delay_test1.c.

10.3 Miscellaneous

10.3.1 Disabling JTAG pins

By default `PB4` and `PB5` are configured as JTAG pins. You can turn them into PIO pins or use them for TWI1 using:

```
#include "mcu.h"

void main (void)
{
    mcu_jtag_disable ();
}
```

10.3.2 Watchdog timer

The watchdog timer is useful for resetting the SAM4S if it hangs in a loop. It is disabled by default but can be enabled using:

```
#include "mcu.h"

void main (void)
{
    mcu_watchdog_enable ();

    while (1)
    {
         /* Do your stuff here.  */

         mcu_watchdog_reset ();
    }
}
```

10.4 I/O model

The mat91lib and mmculib libraries try to use the POSIX I/O model. In general, non-blocking I/O is used so if data is not available, the functions return without waiting, or with a short timeout.

The read/write functions have three arguments:

1. A device handle created by the init function.
2. A pointer to a buffer.
3. The number of bytes to transfer (type `size_t`)

The return value is of type `ssize_t` and is either:

1. The number of bytes transferred if greater than or equal to zero.
2. -1 for error, in which case the error number is stored in `errno`.

Here is an example:

```
char *buffer[4];

ret = adc_read (adc, buffer, sizeof (buffer));
```

10.5 Coding style

For the curious, the GNU C coding style is used. I worked as a compiler engineer for the USA company Cygnus Solutions (taken over by Redhat) on the GNU C compiler (GCC). The first-rule when working as a programmer for a company is to follow the required coding style. My text editor (emacs) is set up to default to this style.

10.6 Under the bonnet

The building is controlled by wacky-racers/src/mat91lib/mat91lib.mk. This is a makefile fragment loaded by wacky-racers/src/mmculib/mmculib.mk. wacky-reacers/src/mat91lib/mat91lib.mk loads other makefile fragments for each peripheral or driver required. It also automatically generates dependency files for the gazillions of other files that are required to make things work.

Chapter 11

Troubleshooting

Troubleshooting can be frustrating when you are tired. You need plenty of gumption and an open mind. Difficult problems are usually a combination of two or more problems. So **do not attempt if you are tired or in a hurry**.

Before you start, you need an A3 printed schmematic, annotated with any changes your group may have made.

11.1 The scientific method

Successful troubleshooting requires application of the scientific method. First you form a hypothesis about the cause of the problem and then you test your hypothesis by making observations. If the observations do not confirm your hypothesis, you need to revise your hypothesis and make more observations.

For example, let's say the IMU does not work. A hypothesis is that the chip has no power, so you test this by measuring the power supply voltage at the chip with an **oscilloscope**. But why not use a multimeter? Well, a multimeter only gives the average value and will not show you that a voltage regulator is oscillating or if there is a lot of noise. If the power supply looks fine, you need another hypothesis, say that the MCU is not outputting I2C signals. You then make observations of the I2C signals, with an oscilloscope, to see if they are changing. If the waveforms seem fine, you might hypothesise that the IMU is not configured for the correct I2C address. So you then check that the acknowledge bit is driven low by the IMU. If there is no acknowledgement, you then check the address that the IMU is configured for and the address sent over the I2C bus. If these match, you might hypothesise that there is a poor electrical connection. So you look at the soldering under a microscope, and so forth.

When forming hypotheses, you need to start with the more likely ones. Hypothesising that a chip is blown is unlikely unless you observed the release of magic smoke, noticed that the chip got very hot, noticed an electrostatic spark, or applied a high voltage, say by putting your PCB on something conductive.

As you get more desperate, you need more outlandish hypotheses about what may have gone wrong. Here, experience helps. However, the key is forming a hypothesis based on previous observations[1].

[1] "Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth." — Sherlock Homes.

11.2 Reporting software errors

A common source of issues is improperly configured software; it is helpful to catch these errors before getting out the oscilloscope and probing the hardware. It is good coding practice to check the return values of library functions for errors. For mmculib and mat91lib library functions, read the function's implementation to determine what value they return on failure. For standard library functions such as `printf`, for example, read the man pages by running `man 3 printf` in the shell or Googling 'man printf(3)'. Generally:

- Functions that return a pointer will return a `NULL` pointer on failure.
- Functions that return an integer will return 0 on success and a negative integer on failure.

You should add a dedicated error LED to your board that you can flash in an error state. For example, you could have a panic routine that flashes the error LED every 400 ms in an infinite loop:

```
static void panic (void)
{
    pio_config_set (LED_ERROR_PIO, PIO_OUTPUT_LOW);
    while(1)
    {
        pio_output_toggle (LED_ERROR_PIO);
        delay_ms (400);
    }
}
```

Then, say you are initialising a PWM module, you should check to see if `pwm_init` returns a `NULL` pointer:

```
pwm_t pwm1 = pwm_init (&pwm1_cfg);
if (! pwm1)
    panic ();
```

You can also use the USB serial interface to report errors. However, try to keep your panic routine simple to minimise the risk of an error occurring during its execution!

It is even better if you can spot software bugs before programming your board. Make sure you compile with the `-Wall` flag and resolve any compiler warnings before programming.

11.3 Oscilloscope

The best tool for debugging an embedded system is an oscilloscope.

1. Use ×10 probes to reduce loading on the circuit.
2. Use the correct probes for the scope so that they can be automatically detected.
3. Compensate the probes by clipping to the probe test signal on the scope and adjusting the variable capacitor in the probe to ensure a square wave without undershoot or overshoot. This is one of the few times you are allowed to use the autoscale button!
4. Check that the probe gains are correct; they should by ×10 for ×10 probes.

5. Hypothesise what you should expect to observe and set up the oscilloscope accordingly. Pushing the auto-set button only works for AC or DC signals and is hopeless for transient signals, such as I2C bus waveforms. For transient signals, use normal mode triggering.

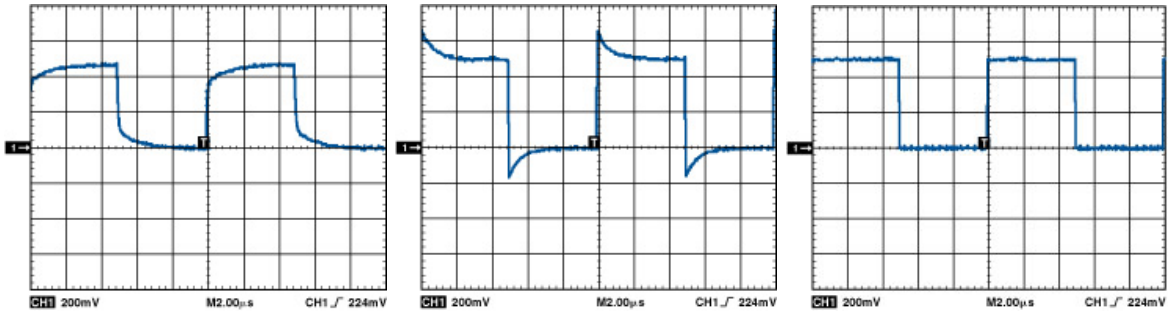


Figure 11.1: Oscilloscope probe compensation: under, over, and proper. From http://www.analog.com/library/analogdialogue/archives/41-03/time_domain.html.

11.3.1 Normal mode

This mode is for measuring transient or non-repetitive signals, such a serial bus waveforms. It will only refresh the display if a trigger event is detected. You need to adjust the trigger level.

It is useful to increase the trigger holdoff to prevent triggering on multiple edges of a waveform. By default, it is set to a ridiculously small value.

11.3.2 Auto mode

This mode is for measuring AC or DC signals. It is like normal mode but if it does not detect a trigger it will automatically refresh the display.

11.4 SAM4S not detected by OpenOCD

If a program has not been loaded:

1. Check orientation of the SAM4S (the large circle does **not** mark pin 1).
2. Check soldering of the SAM4S pins under a microscope. Giving each pin a push with a sharp spike can reveal a poorly soldered joint.
3. Check 3.3 V and 1.2 V power rails.
4. Check the serial wire debug (SWD) signals, see debugging serial wire debug (SWD).

If a program has been loaded:

1. Check the crystal oscillator.
2. Erase flash memory.
3. Re-enable booting from flash memory.
4. Reprogram the SAM4S with an LED flash program.

11.4.1 Erasing flash memory

1. Connect the ERASE pin to 3.3 V.
2. Re-start OpenOCD.
3. Re-enable booting from flash memory.

11.4.2 Debugging serial wire debug (SWD)

`OpenOCD` periodically polls to see if the SAM4S is alive. These occur every 100 ms, see Figure 11.2. If the SAM4S responds, the waveform seen in Figure 11.3 should be observed on the SWD pin.

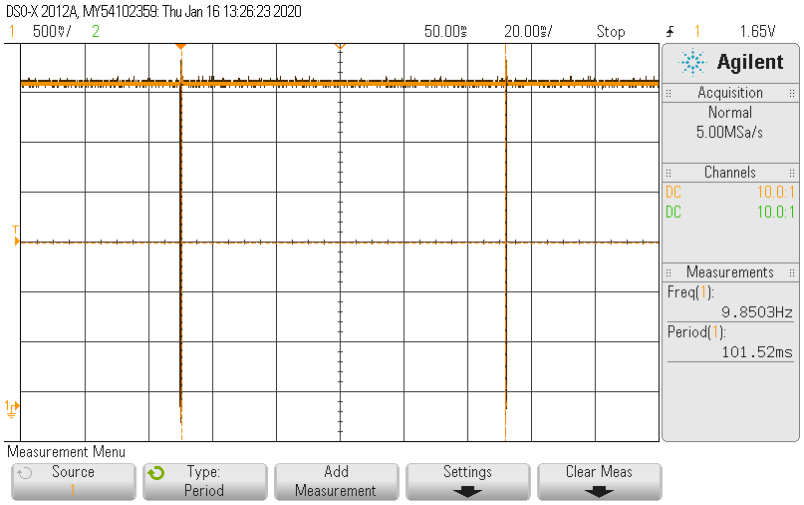


Figure 11.2: OpenOCD polls every 100 ms while connected.

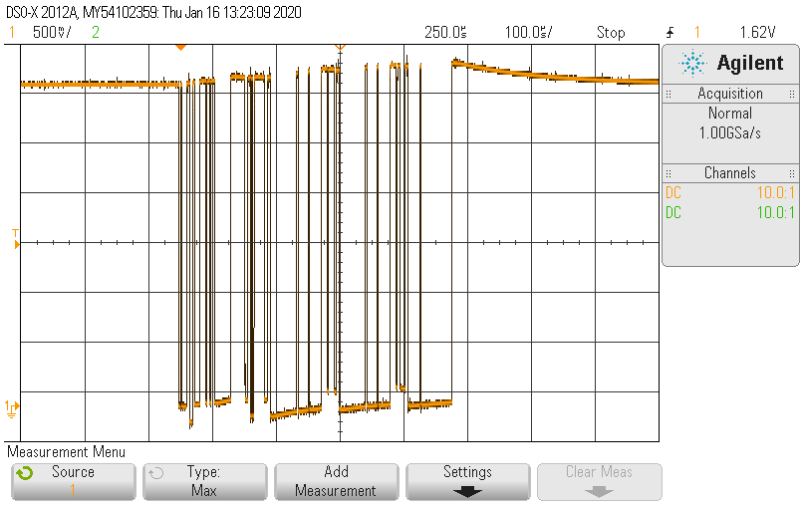


Figure 11.3: SWD signal when there is a response from the SAM4S.

OpenOCD polls less often when it does not get a response. The period increases to 6.3 s.

11.4.3 Checking the crystal oscillator

When the SAM4S is first powered up it uses an internal RC oscillator. However, when a program is loaded, the code before main is called switches the SAM4S to use the main oscillator that uses the external crystal. If there is a problem with the crystal oscillator the SAM4S will not run since it has no clock. Moreover, `OpenOCD` will then fail to communicate with the SAM4S.

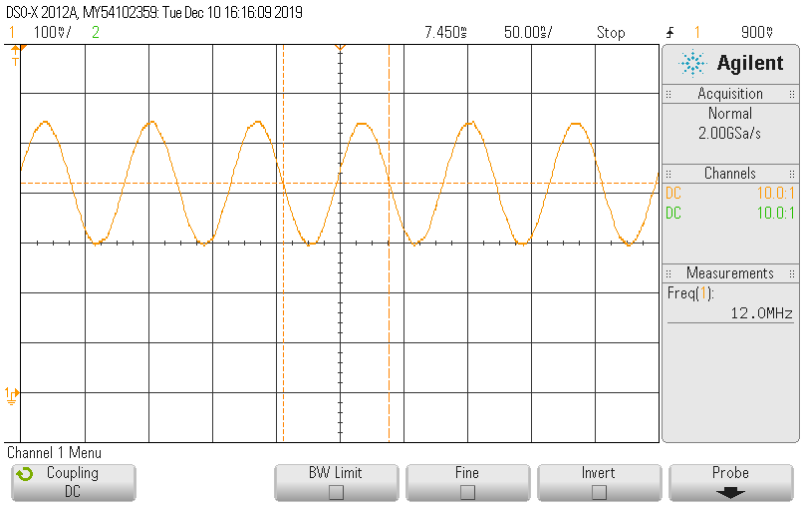


Figure 11.4: 12 MHz clock sine wave measured on the XOUT pin.

Add scope picture of CPU clock signals when not working

Figure 11.5: Oscillator waveform when not working.

1. Connect a scope pin to the XOUT pin. A 12 MHz sinewave should be visible, see Figure 11.4.
2. If there is no sinewave:
   (a) Measure the voltage on the VDDPLL pin of the MCU with a scope. This should be 3.3 V to power the oscillator.
   (b) Check the bypass capacitors across the crystal. These should be approx. 20 pF.

11.4.4 Checking the clock

The SAM4s has multiple clock sources:

1. Internal fast RC oscillator (this is selected when the SAM4S is first ever used).
2. Internal slow RC oscillator (this can be selected to save power).
3. Main oscillator using external crystal.

The SAM4S uses a phase-locked-loop (PLL) to multiply the frequency of the clock source to provide the CPU clock. Sometimes the PLL will run without a clock source and thus will generate an unexpected frequency.

The clock frequency can be checked by connecting a scope to a peripheral pin that generates a clock, e.g., PWM, SCK, TXD, and programming the peripheral.

11.5 USB serial

USB is a complicated protocol and there are many possibilities why it does not work.

- Check that the termination resistors are 27 ohms.
- Check the SAM4S is running at the correct frequency of 96 MHz[2]. Check the SAM4S XIN pin with an oscilloscope for a 12 MHz sinusoid. Note, the oscillator is not enabled unless a program has been loaded and running. If a 12 MHz signal is not found, check the MCU solder connections under a microscope. Also check that the VDDPLL pin has 3.3 V.

If the USB serial connection drops characters:

- Add a delay before sending data. This is because the driver takes a while to set up the connection after the USB cable is plugged in. If you try to send some data during this time, the data gets stored into a ring buffer for later transmission. However, the ring buffer is not large and once it is filled, the USB serial driver will drop characters.

11.6 Motors

For the motors to work:

1. The H-bridge must be correctly configured.
2. The PWM signals must be correctly generated.

11.6.1 Testing the H-bridge

1. The `nFAULT` pin should be high. Note, this is an open-drain pin and requires a pullup resistor to 3V3 to make it work. Without this resistor, this pin will always read low, fault or no fault. If this is connected to the SAM4S, it will be pulled up by default.
2. The `nSLEEP` pin needs to be high to enable the chip.
3. Check that the capacitor connected to the `INT` pin is 2.2 μF and not 2.2 nF.
4. Check the `AIN1`, `AIN2`, `BIN1`, `BIN2` pins. If you see DC 2 V, this is due to the signal not being configured as an output on the SAM4S; the internal pullup of the SAM4S forms a voltage divider with the internal pulldown of the H-bridge chip.
5. Check that `AINSENSE` and `BINSENSE` are connected directly to ground or to ground via a small resistor if you want current limiting.

11.6.2 Debugging PWM

If PWM does not work:

1. Check the SAM4S pin since not every pin can be a PWM signal.
2. Check the definition in the configuration file `target.h`

If the PWM frequency is wrong:

[2]The SAM4S has two PLLs so it is possible to clock the CPU at other frequencies.

1. Check the clock frequency, see checking the clock.
2. Check your program.

If the PWM duty is wrong:

1. Check your program. There are two ways to set the duty cycle with mat91lib's PWM module: (1) setting the `duty` field of the `pwm_cfg_t` struct using the `PWM_DUTY_DIVISOR` macro and passing the duty cycle as a percentage or (2) setting the `duty_ppt` field of the `pwm_cfg_t` struct as an integer in parts per thousand (e.g., 1000 = 100% duty cycle; 50 = 5% duty cycle). Setting the latter field will override the former. See the definition of `pwm_cfg_t` in `wacky-racers/src/mat91lib/pwm/pwm.h`.

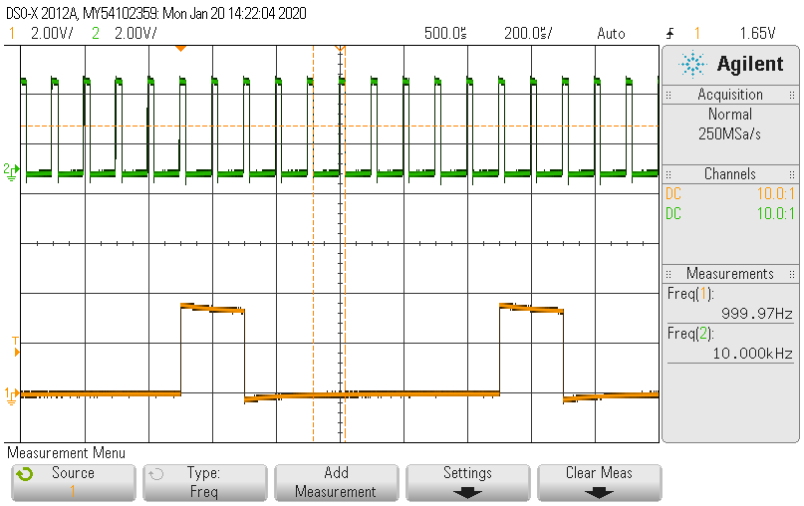


Figure 11.6: Two PWM signals at 1 kHz and 10 kHz.

11.7 IMU

For the IMU to work:

1. The IMU chip must be correctly configured.
2. The I2C bus must be working.

11.7.1 IMU checking

1. Check the I2C bus (see debugging I2C).
2. Check the auxiliary I2C bus signals are not connected on the IMU (otherwise the magnetometer will not respond).
3. Check `nCS` on the IMU is pulled high.
4. Check `FSYNC` on the IMU is pulled low.
5. Try pressing on a side of the IMU with a fingernail to see if it starts working; you might have a dry joint. Otherwise look under the microscope at the pins of the IMU and touch them up with a soldering iron if necessary.

6. Check that you are using the correct I2C address corresponding the the state of the AD0 pin (see datasheet).

11.7.2 Debugging I2C

1. I2C/TWI requires external pull-up resistors for the clock (TCK) and data (TDA) signals. Use a scope to look at the signals `TWCK/SCL` and `TWD/SDA`. If the rise times are too slow so that the high voltage level is not 3.3 V, the resistors are too large.
2. TWI channel 1 (TWI1) uses `PB4` and `PB5`. However, these are configured on boot as JTAG pins. You can disable this, see disabling JTAG pins.

Add scope picture of I2C signals

Figure 11.7: I2c signals with acknowledgement.

Add scope picture of I2C signals with no acknowledge

Figure 11.8: I2c signals with no acknowledgement.

11.8 NRF24L01+ radio

For the radio to work you need:

1. No ground planes near the antenna[3].
2. A working SPI interface.
3. The correct channel and address.
4. A well filtered power supply for the radio, see Figure 2.1.
5. A channel with little interference (note, some channels are shared with WiFi and Bluetooth and may be less reliable).

If nothing works, check:

1. For transmission the CE pin should be low; for reception the CE pin should be high.
2. Check that the SPI clock SCK, MOSI, and MISO signals are driven when the radio is configured (note, the MISO signal is tristate when CS is high).
3. Check that the SPI CS signal is driven low for each transmitted byte.

If the radio transmits but not receives, check:

1. The IRQ pin is driven low (this indicates that a packet has been received).
2. The power supply. The radio requires a well filtered power supply otherwise the range will be limited on reception. Preferably, the device should have its own 3V3 regulator with a low-pass RC filter comprised of a series resistor and large capacitor (say 22 μF) or a low-pass LC filter made with a ferrite bead and capacitor.

[3] You might have to mount the radio vertically.

Note, by default the radio waits for an auto-acknowledgement from the receiver device. This acknowledgement is performed in hardware. If no acknowledgement is received, it retries for up to 15 times. The auto-acknowledgement and number of retries can be configured in software.

11.8.1 Checking the radio power supply

Add scope picture of expected radio power supply voltage

Figure 11.9: Waveform showing radio power supply voltage.

11.8.2 Debugging SPI

Check:

1. The device chip select signal is driven low.
2. The `SCK` signal toggles while the chip select is low.
3. The `MOSI` signal does something while the chip select is low.

The `MISO` signal is usually tri-state and is driven by the device when the chip select is low.

The SPI standard is rather loose and there are four modes to confuse the unwary; these are due to two clock polarity modes and two clock phase modes. When using the wrong mode, the read data can be out by a bit or unreliable.

Add scope picture of SPI signals

Figure 11.10: SPI signals.

11.9 Other problems

1. `OpenOCD` does not run. The most common problem is that the USB permissions are not correct[4].
2. A program does not correctly build after a file has been changed. Every now and then the file timestamps are incorrect (this is a common problem with network drives due to a skew in the clocks) and make will not correctly rebuild a file. Running **make clean** will remove the existing dependency and object files so you can start afresh.
3. A program hangs. This can be observed by an LED no longer flashing. There are a number of reasons:
   (a) The program is stuck in an infinite loop. Note, an embedded system should only have an infinite loop for the main loop; all other loops should have a timeout condition.
   (b) The program has crashed trying to access invalid memory. Usually this is due to buffer overflow or dereferencing uninitialised pointers (say by not calling `radio_init`). Try running `make debug`. This will start the debugger and attach to your MCU. If the debugger says that your program has stopped in `_hardfault_handler`, then your program

[4]On Linux this is controlled by udev.

is likely to have accessed invalid memory. Use the `bt` command to print a stack trace to see how your program went astray.

(c) You are printing output using USB but a terminal program is not running.

4. The output of the 3V3 regulator works fine when powered from USB but gives 6 V when powered from a 7.2 V battery. Check that the 7.2 V is not connected directly to a SAM4S pin (say for battery monitoring) since this will cause an ESD protection diode inside the SAM4S to conduct.

5. Sometimes the program works but other times it does not The likely causes are:

   (a) You have an uninitialised variable—check your code.

   (b) You have a poor solder connection for a MCU pin—check with microscope.

   (c) You have a race condition—this is unlikely unless you are using interrupts.

Chapter 12

Rework

Flux is your friend when doing rework. However, you need to remove it from your board when finished, see flux removal. You should check all rework with a magnifier and look for solder bridges.

12.1 Removing resistors and capacitors

1. Apply flux from syringe to pads.
2. Use tweezer soldering iron.

12.2 Removing larger parts

1. Apply flux from syringe to pads.
2. Use a grunty soldering iron. **Do not push hard with a small soldering iron since this will bend the tips.**

12.3 Removing MCU

1. Only remove as last resort.
2. Apply flux from syringe to all pins.
3. Get Scott to show you how to use the hot-air gun. The key is to choose a head that matches the size of the MCU and getting the temperature correct.
4. Be careful not to lift pads or tracks.

12.4 Adding a wire

1. For a signal trace, choose the very fine wire.
2. Cut to length and strip insulation from both ends. The second end is a lot harder!
3. Bend wire to shape and tape down to the PCB. Note, you can run this wire through vias.

4. Apply flux from syringe to both pads.
5. Use small soldering iron to solder wires in place.

12.5 Flux removal

1. Scrub board with using flux-cleaner and brush.
2. Clean board with a towel.
3. Wash board with a little bit of IPA.
4. Clean board with a towel.

Show photos of clean board and dirty board

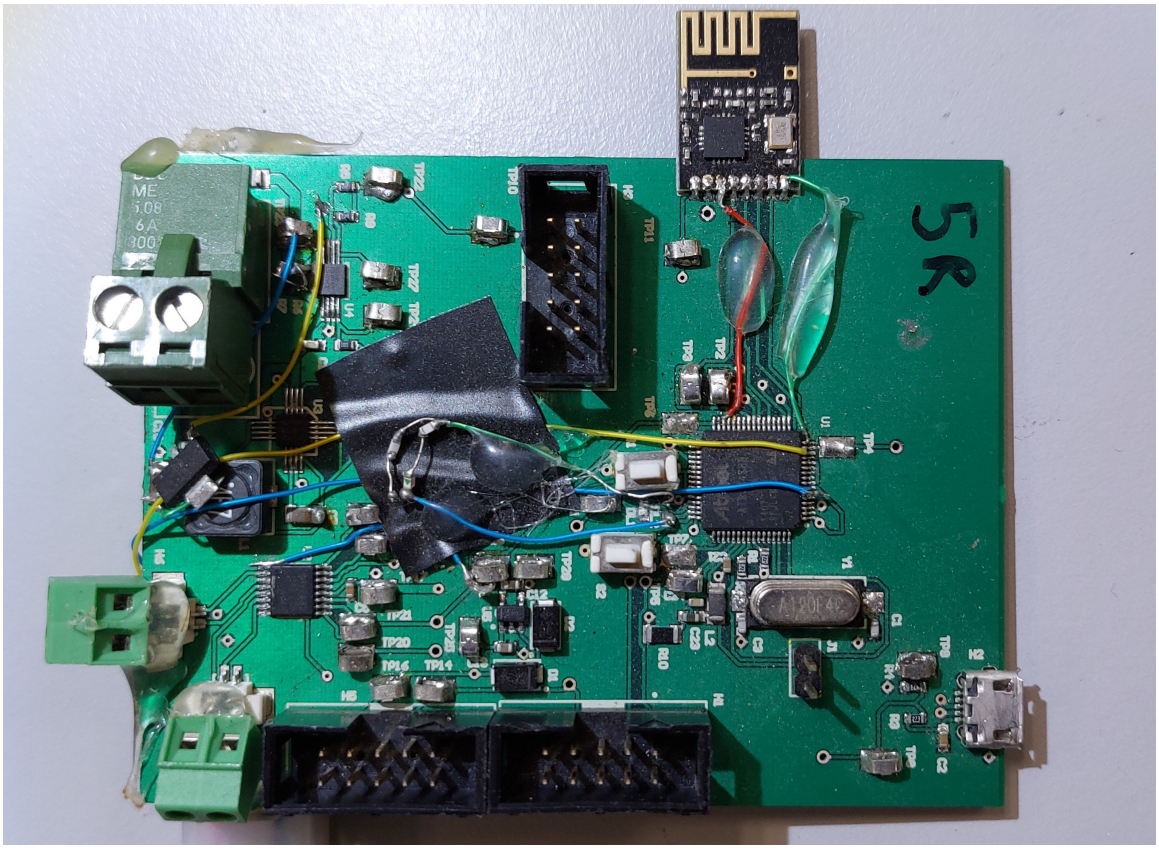


Figure 12.1: Example of a PCB where not much care was taken with the schematic resulting in a lot of rework. The board worked!

Chapter 13

Debugging

Your SAM4S can be debugged using OpenOCD in conjunction with GDB. However, debugging embedded systems is difficult due to asynchronous behaviour. Moreover, when optimising, a compiler will rearrange (and even remove) code and hold variables in registers.

13.1 Command-line debugging with GDB

If `OpenOCD` is running, a running program can be debugged using:

```
$ make debug
```

This starts `GDB` and attaches to the SAM4S. `GDB` is a command line debugger but there are many GUI programs that will control it, for example, `vscode` and `geany` have plugins.

`GDB` allows you to inspect the CPU registers, memory, set breakpoints, set watchpoints, and much more.

You can reset your program using the GDB `jump reset` command. However, this does not reset the peripherals as with a power-on reset.

The backtrace, `bt`, command is useful to show the function call stack.

A handy overview of GDB's CLI commands can be found at: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf.

Appendix A

OpenOCD

The Open On-Chip Debugger (OpenOCD) is an open-source on-chip debugging, in-system programming, and boundary-scan testing tool. It is able to communicate with various ARM and MIPS microprocessors via JTAG or SWD. It works with a number of different JTAG or SWD interfaces/programmers. User interaction can be achieved via telnet or the GNU debugger (GDB).

A.1 Configuration files

OpenOCD needs a configuration file to specify the interface (USB or parallel port) and the target system. Unfortunately, these change with every new release of OpenOCD.

A.2 Running OpenOCD

OpenOCD runs as a daemon program in the background and can be controlled from other programs using TCP/IP sockets. This means that you can remotely debug from another computer. The socket ports it uses are specified in the OpenOCD configuration file supplied when it starts. By default, OpenOCD uses port 3333 for GDB and port 4444 for general interaction using Telnet.

A.2.1 Communicating with OpenOCD using telnet

If OpenOCD is running, commands can be sent to it using telnet. For example,

```
$ telnet localhost:4444
> monitor at91sam4s info
```

A.2.2 Communicating with OpenOCD using GDB

If OpenOCD is running, commands can be set to it with GDB. There are two steps: connecting to OpenOCD with the target remote command, and then sending a command with the GDB monitor command. For example,

```
$ gdb
(gdb) target remote localhost:3333
(gdb) monitor flash info 0
```

A.3 OpenOCD commands

All the gory OpenOCD details can be found in the OpenOCD manual. If you are getting strange errors see OpenOCD errors.

A.4 Flash programming

OpenOCD can program the flash program memory from a binary file.

A.5 Errors

See OpenOCD errors.

Appendix B

Git

To properly use git you should commit and push often. The smaller the changes and the more often you make per commit, the smaller the chance of the dreaded merge conflict.

B.1 Typical workflow

1. Edit file
2. Save changes
3. Check differences

   ```
   $ git diff
   ```

4. Stage files you want to commit

   ```
   $ git add [list-of-modified-files]
   ```

5. Commit changes

   ```
   $ git commit -m "[Commit message]"
   ```

   Note, you should commit at least every 15 min, preferably when you have made a single functional change. Ideally each commit should be self-contained. **Note, do not add binary files (.o) etc.** These will make merging even more miserable.

6. Push changes to server

   ```
   $ git push
   ```

   The more often you push, the lower the chance that you will get a merge conflict.

B.2 Diff, status, blame, log

The diff command is useful to determine what changes you made.

The status command says which files have been modified and what you should do, say when you get a merge conflict.

The blame command is useful to determine who authored each line of code.

The log command shows all the previous commit messages.

B.3 Pulling from upstream

To be able to get updates if the template project is modified you will need to:

```
$ cd wacky-racers
$ git remote add upstream https://eng-git.canterbury.ac.nz/wacky-racers/wacky-racers.git
```

Again if you do not want to manually enter your password (and have ssh-keys uploaded) you can use:

```
$ cd wacky-racers
$ git remote add upstream
↪  git@eng-git.canterbury.ac.nz:wacky-racers/wacky-racers.git
```

Once you have defined the upstream source, to get the updates from the main repository use:

```
$ git pull upstream master
```

If you enter the wrong URL by mistake, you can list the remote servers and edit the dodgy entry:

```
$ git remote -v
$ git remote set-url upstream [new-url]
```

Note, **origin** refers to your group project and **upstream** refers to the template project that origin was forked from.

B.4 Merging

The bane of all version control programs is dealing with a merge conflict. You can reduce the chance of this happening by sticking to these precautions:

- Keep hat and racer code in separate directories.
- Limit commits to single changes: if you make changes to a bunch of different files (e.g., changes to both the hat and racer code), break these up into separate commits by selectively staging files with the `git add` command (see Section B.1).
- Try to avoid having more than one developer work on the same code at the same time on different machines.

If you get a message such as:

```
From https://eng-git.canterbury.ac.nz/wacky-racers/wacky-racers
 * branch            master     -> FETCH_HEAD
error: Your local changes to the following files would be overwritten by merge:
    src/test-apps/imu_test1/imu_test1.c
Please, commit your changes or stash them before you can merge.
```

what you should do is:

```
$ git stash
$ git pull
$ git stash pop
# You may now have a merge error and will have to edit the problem
```

```
# file, in this case imu_test1.c. Once the file has been fixed:
$ git add src/test-apps/imu_test1/imu_test1.c
$ git commit -m "Fix merge"
```

A handy command is `git diff --check`, which will tell you if there are any leftover merge conflict markers in your files. You should run this before committing after resolving a merge conflict. If you are stuck in a messy merge conflict `git merge --abort` will reset your local repository back to the state it was in before the action that caused the conflict.

A common issue that can occur when collaborating with multiple developers on a single Git repo is when you attempt to push local commits but someone else has pushed commits that you haven't pulled into your local repo. You will get an error like:

```
error: failed to push some refs to
    'git@eng-git.canterbury.ac.nz:wacky-racers/wacky-racers.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

If you run `git pull` like the message suggests you will be thrown into a text editor to type a merge comment. To avoid this, run `git pull --rebase` (`git pull -r` for short) to pull from the remote and move your local commits on top of the new remote commits (i.e. 'rebase' them). You can then do a `git push` as normal. As long as the remote commits do not conflict with your local commits there will be no merge conflicts or merge commit.

For cases where you do have to write a merge commit, the choice of editor is controlled by an environment variable `GIT_EDITOR`. On the ECE computers this defaults to emacs[1] on Linux. You can changed this by adding a line such as the following to the `.bash_profile` file in your home directory.

```
$ export GIT_EDITOR=geany
```

[1] To exit emacs type Ctrl-x Ctrl-c, to exit vi or vim press Esc and type `:q!`.

Appendix C

Sleeping

Sleeping a MCU is important for embedded systems applications to prolong battery life.

C.1 Dynamic power consumption

The dynamic power consumption for CMOS is

$$P = fCV^2, \tag{C.1}$$

where f is the clock frequency, C is the total switched capacitance, and V is the power supply voltage. It can be difficult to reduce V for a MCU and so the power consumption can only be reduced by lowering the clock frequency, f, and/or shutting down parts of the MCU to reduce C.

C.2 Slowing the CPU clock

The SAM4S, like many MCUs, can be clocked from a number of sources. For example, it has a slow clock generated by an RC oscillator with a frequency of 32768 Hz.

With the mat91lib library the slow clock can be selected using:

```
mcu_select_slowclock ();
```

Warning, switching to a slow CPU clock will cause havoc with OpenOCD. You might not be able to load a new program. In this case, you will need to erase flash memory.

C.3 Disabling the CPU clock

Further power reduction can be achieved by disabling the CPU clock. In effect, this reduces C in (C.1). The clock is disabled using the ARM WFI (wait for interrupt) instruction. The clock remains disabled until an interrupt occurs. The WFI instruction is executed by calling the mat91lib `cpu_wfi ()` function, defined as:

```
static inline void
cpu_wfi (void)
{
```

```
    __asm__ ("\twfi");
}
```

Warning, before executing the WFI instruction, it is necessary to enable an interrupt to wake the CPU.

Warning, disabling the CPU clock will cause havoc with OpenOCD.

C.4 Disabling peripherals

Further power reduction can be achieved by shutting down peripherals that are not required. This also reduces C in (C.1). The drivers in mat91lib have a shutdown function, e.g., `spi_shutdown ()`.

C.5 Current measurement

Measuring the current to determine power consumption is not straightforward. Usually, the voltage drop across a known resistance is measured. For normal operation, this resistance needs a low value otherwise there will be significant voltage drop and the MCU will not run. When sleeping, a much larger resistor is required so that the voltage drop can be measured (all going well the MCU will only take a few microamps when sleeping). One approach is to use two voltage drop resistors connected in parallel when the MCU is running normally, and to switch out the low value one when the MCU is sleeping. There are special current measuring devices that dynamically vary the voltage drop resistor.

Appendix D

EMC

Electromagnetic emission (EMI) occurs with changing electric and magnetic fields. A good PCB design should have good electromagnetic compatibility (EMC); this means that it has low EMI and that is has low susceptibility to changing electric and magnetic fields.

D.1 Electromagnetic coupling

It is worth recapping the fundamental equations. A changing aggressor current, $i_a(t)$, flowing around a loop induces a voltage[1] in a victim loop according to Faraday's law,

$$v_v(t) = M\frac{\mathrm{d}i_a(t)}{\mathrm{d}t}, \tag{D.1}$$

where M is the mutual inductance. The mutual inductance depends on the areas of the aggressor and victim loops and their orientation. It is minimised by using microstrips.

A changing aggressor voltage induces a current in a victim circuit according to

$$i_v(t) = C\frac{\mathrm{d}v_a(t)}{\mathrm{d}t}, \tag{D.2}$$

where C is the mutual capacitance that depends on the separation between the traces and the distance the traces run beside each other. From Ohm's law, the victim current will produce a voltage $i_v(t)R$ across a resistance R. Thus low resistance circuits are more immune to capacitive coupling.

[1] This voltage magically appears around a loop and does not obey Kirchhoff's voltage law. It will mostly 'appear' across the highest resistance in the loop.

Appendix E

PIO pins

There are five types of PIO driver, see Table E.1, with different drive strengths. The SPI signals are designed for 70 MHz operation and require greater drive strengths. Each pin[1] has a series resistor for on-die-termination (ODT) to reduce reflections. The maximum current applies for DC operation; the drivers can momentarily provide much more current.

[1] Except the USB pins; these need external 27 ohm series termination resistors.

Pin	Pin group	ODT	Imax (mA)	fmax (MHz)	ADC	Speciality
PA0	4	18	2	58		
PA1	4	18	2	58		
PA2	4	18	2	58		
PA3	4	18	2	58		TWD0 (TWI0)
PA4	2	36	2	46		TWCK0 (TWI0)
PA5	2	36	2	46		
PA6	2	36	2	46		
PA7	2	36	2	46		
PA8	2	36	2	46		
PA9	2	36	2	46		
PA10	2	36	2	46		
PA11	2	36	2	46		
PA12	3	36	2	70		MISO (SPI)
PA13	3	36	2	70		MOSI (SPI)
PA14	1	36	4	70		SPCK (SPI)
PA15	2	36	2	46		TF (SSC)
PA16	2	36	2	46		TK (SSC)
PA17	2	36	2	46	Y	TD (SSC)
PA18	2	36	2	46	Y	RD (SSC)
PA19	2	36	2	46	Y	RK (SSC)
PA20	2	36	2	46	Y	RF (SSC)
PA21	2	36	2	46	Y	
PA22	2	36	2	46	Y	
PA23	2	36	2	46		
PA24	2	36	2	46		
PA25	2	36	2	46		
PA26	3	36	2	70		MCDA2 (multimedia)
PA27	3	36	2	70		MCDA3 (multimedia)
PA28	3	36	2	70		MCCDA (multimedia)
PA29	1	36	4	70		MCCK (multimedia)
PA30	3	36	2	70		MCDA0 (multimedia)
PA31	3	36	2	70		MCDA1 (multimedia)
PB0	2	36	2	46	Y	
PB1	2	36	2	46	Y	
PB2	2	36	2	46	Y	
PB3	2	36	2	46	Y	
PB4	2	36	2	46		TDI (JTAG)
PB5	2	36	2	46		TDO (JTAG)
PB6	2	36	2	46		TMS (JTAG)
PB7	2	36	2	46		TCK (JTAG)
PB8	2	36	2	46		XOUT (crystal)
PB9	2	36	2	46		XIN (crystal)
PB10	5	0	30	25		DDM (USB, needs series R)
PB11	5	0	30	25		DDP (USB, needs series R)
PB12	2	36	2	46		ERASE
PB13	2	36	2	46		DAC0
PB14	2	36	2	46		DAC1

Table E.1: SAM4S PIO pins. ODT is the on-die termination resistance. Imax is the maximum steady state current.