A Programmable Computer Fan Controller

Liam McSherry

Higher National Diploma in Electrical Engineering

2018

• PAGE INTENTIONALLY LEFT BLANK •

A Programmable Computer Fan Controller

by Liam McSherry

Submitted in partial fulfilment of the requirements for a
HIGHER NATIONAL DIPLOMA IN ELECTRICAL ENGINEERING

Supervised by
Martin McLean

Department of Engineering
Edinburgh College

2018

A Programmable Computer Fan Controller

2017–2018 © Liam McSherry

A report, submitted in partial fulfilment of the requirements for a Higher National Diploma in Electrical Engineering, on the theory of operation, design, and construction of an electronic controller for computer fans.

304 pages and approximately 82,100 words.

Set in Pablo Impallari's Caslon and Franklin.

Printed in Hertfordshire by Mixam UK Ltd.

Abstract

This report seeks to determine the theory behind the control of typical computer fans and apply this in the design and construction of a prototype for an electronic programmable fan controller, and to then apply the knowledge and observations gained in doing so to make recommendations for a design for a fan controller suitable for commercial and volume production.

The types of fan typical in computer systems vary significantly, with the capacity to receive a control signal or report status information not guaranteed. As such, a fan controller must employ a number of techniques to control all typical varieties of fan. This report initially explores three techniques—use of a standard control signal for those fans which support it, modulation of the fan's power supply so as to reduce the average voltage supplied to the fan, and use of a filter arrangement with modulation so as to reduce the absolute voltage supplied to the fan.

Additionally, this report considers and demonstrates what is necessary to have a fan controller be capable of receiving instruction from a user through a software interface on a host computer system—a capacity which, in conjunction with a set production cost target of £40 per unit and the ability to mount the controller in a standard computer chassis bay, would be likely to offer an improvement over fan controllers currently on the market.

This report then makes wide recommendations for a production design based on the research and the knowledge gained by prototyping. In particular, this report recommends that a production design use both a standard control signal and the modulation of the supply to control a fan, that the design be connected to a host computer using the Universal Serial Bus (USB), and that the design be suitable for mounting in a standard 5.25" bay of a typical computer chassis.

Acknowledgements

Much of the testing and prototyping done as part of this project was done using tools, equipment, and workshops at Edinburgh Napier University, and so I extend my thanks to Jay Hoy for his help and accommodation in making these resources available, without which it may not have been possible to carry out this project to the same standard.

Table of Partitions

Table of Appendices

Requirements Specification

1. Brief

1.1 Context

The primary purpose of a computer fan is to ensure that the components in a computer system remain within safe temperature ranges, and although doing so would fulfil that purpose, it is not desirable to constantly operate computer fans at full speed—the constant high-speed operation could cause unnecessary wear on the fan, or could produce excessive levels of noise. Therefore, it is desirable that the speed of a computer fan be controlled to suit dynamic conditions.

1.2 Aim

The aim of the project is to produce a prototype computer fan controller which is readily convertible, with minimal further work, to a design which is suitable for commercial and volume production. Included with this prototype must be the firmware for the controller and the software necessary for the host computer to interface with and drive the controller.

For the purposes of fulfilling this aim, there are the following objectives:

1.2.1 To develop a product which requires minimal or no in-depth technical knowledge to use, and which can be used with minimal prior instruction.

1.2.2 To attain the lowest practical per-unit cost without excessive detriment to quality, reliability, or safety; and in any case to not exceed the per-unit cost maximum set out in requirement 2.1.8.

1.2.3 To deliver the prototype in line with the proposed schedule.

1.2.4 To develop the prototype at a cost of £600 or less.

1.2.5 To produce, to a high standard, all materials necessary for reproduction of the prototype; and to include the materials herein.

1.2.6 To produce a robust and, where possible, automated suite of tests for each aspect (hardware, firmware, and software) of the prototype.

1.2.7 To have regard to the ease of manufacturing of the product, and in doing so to identify and implement accepted "design for manufacturing" techniques and practices (DFM).

1.2.8 To identify and comply with all relevant standards and applicable law, with particular regard to standards and law concerned with health and safety and electromagnetic compatibility.

2. Requirements

This chapter sets out the requirements for the project. It is divided into three sections for clarity: hardware, covering the physical controller; firmware, relating to the software operating the controller hardware; and software, for the software that executes on the host computer to interface with the controller hardware and firmware and perform related functions.

2.1 Hardware requirements

The requirements for the fan controller hardware are as follows:

2.1.1 The controller must be able to control the speed of all types of fan commonly used within computer systems.

2.1.2 The controller must support the control of at least four fans.

2.1.3 The controller must be suitable for mounting in a typical desktop- or tower-style computer chassis.

2.1.4 The controller must not require the use of a power supply separate from the primary power supply for the host computer.

2.1.5 The controller must interface with the host computer using standard and widely-available means or methods.

2.1.6 The controller must have means of acquiring relevant data, such as the temperature of its environment.

2.1.7 The controller must comply with all applicable health and safety law.

2.1.8 The cost of producing the controller in reasonable commercial volume should not exceed £40 per unit.

2.2 Firmware requirements

The requirements for the fan controller firmware are as follows:

2.2.1 The firmware must be able to control and (where possible) monitor and report to the host computer the speed of the fans to which the controller is connected.

2.2.2 The firmware must be able to interface with any transducers or other equipment included by virtue of requirement 2.1.6, and process and act upon the data thereby acquired; and must be able to report the data, before or after its processing (as appropriate), to the host computer.

2.2.3 The firmware must be able to detect the failure of any connected fans, and must be able to report such failures to the host computer.

2.2.4 The firmware must make such provision necessary to support the means or method selected to fulfil requirement 2.1.5; and should support the in-circuit updating of the firmware by that means or method.

2.2.5 The firmware must support the storage of configuration data.

2.2.6 The firmware must support complex configurations, such as mapping temperature data to fan speed by a user-provided function.

2.3 Software requirements

The requirements for the host computer software are as follows:

2.3.1 The software must be able to display, in real time, the data reported in fulfilment of requirements 2.2.1 through 2.2.3.

2.3.2 The software must be able to provide configuration data based on user input to the fan controller, including data for complex configurations.

2.3.3 The software should support the updating of the fan controller firmware.

3. Schedule

3.1 Proposed schedule

The project is anticipated to take 24 weeks to complete, comprising:

- **7 weeks for fundamental research**
  The research necessary to produce the design specification. For example, this would include research into the relevant standards, required hardware subsystems such as power delivery and controller–host communication, and the operating system interfaces and systems with which the host computer software will be required to interact.

- **10 weeks for hardware, firmware, and software development**
  The producing of the design specification; the simulation and design of the hardware; and the development of the firmware and software and test suites therefor. The prototype hardware must be ready for manufacture by the end of the first three weeks of this period. Additional development, testing, and refinement of the firmware and software is done during manufacture.

- **4 weeks for project evaluation, proofing, editing, etc.**
  The project is evaluated, any final additions are made to the report, editing and proofreading which is necessary is done, and the report is printed.

- **3 weeks for slippage**
  Additional time allocated to account for unforeseen delays.

The project must be completed on or before 20th April 2018. To be completed on this date, provided that the proposed schedule is adhered to entirely, the project must begin no later than 3rd November 2017.

A Gantt chart illustrating the proposed schedule is included in Appendix A1.

3.2 Actual schedule

It is expected that, once the details of each aspect of the project become clearer, changes to the schedule will need to be made. Appendix A2 details the revisions made to the proposed schedule throughout the project.

3.3 Progress log

To record the progress made and work undertaken in relation to the project, a progress log was kept. The completed log is contained in Appendix B.

Research and Theory

4. Summary of findings

This chapter summarises the findings made during research for the project. More detailed information is provided in the following chapters, which are referenced where appropriate in this summary.

4.1 Hardware findings

There were the following findings in relation to the controller hardware:

- There are four principal types of computer fan, three of which use inter-compatible variants of the same connector. The fourth type is not common enough to warrant support (see chapter 5.1).
- Fans are likely to all use brushless D.C. motors (see chapter 5.2).
- Fans are controllable by reduction of the voltage supplied to them. In the case of 2- and 4-pin fans, this can be achieved without impact to torque by reduction of the average voltage with PWM (see chapter 6). In the case of 3-pin fans, however, the absolute voltage must be reduced so as not to interfere with the operation of sensors on the fan motor (see chapters 5.3 and 5.4).
- Fans take a 12 V input, and can draw up to 18 W continuously (26.4 W during starting). However, most modern fans have considerably lower power requirements (see chapter 5.5).
- The most appropriate form factor for the controller is one derived from that

used for 5.25” optical media readers, enabling controller mounting in 5.25” drive bays provided in most computer chassis (see chapter 7).

- The most appropriate host–controller interface is USB (see chapter 9).
- The computer fans are to be supplied through a “2 × 3” PCI Express auxiliary power connector, and lower-power aspects of the controller from the USB power connection from the host–controller interface (see chapter 8.1).
- The majority of fans consume considerably less than the 18 W permitted by the specification for 4-pin fans, and so the 75 W provided by the auxiliary power connector should be more than sufficient for four fans. Specifically, all fans surveyed consumed less than approximately 6.5 W (see chapter 8.2, in the section on power supply margins).
- The absolute voltage reduction required for the control of 3-pin fan types can be attained by use of a so-called “PWM DAC,” where the controller feeds a PWM signal into a filter, which removes high-frequency components to produce a largely steady voltage with magnitude proportional to the duty cycle of the PWM signal (see chapter 8.2).
- Power delivery to a microcontroller is anticipated to be uncomplicated, and it is expected that a simple linear regulator will be sufficient (see chapter 8.3).
- A device in the Silicon Labs EFM32WG family of microcontrollers is to be used as the processor in the fan controller (see chapter 10.1).
- For the PWM DAC, the LC filter is to use an inductance of approximately 470 μH and a capacitance of 4.7 μF; the snubber circuit is to use the Nexperia TDZ12J Zener diode with a current-limiting resistance of at least 1078 ohms; and the control transistor is to be the Fairchild FDMS7682 (see chapter 10.2).

4.2 Firmware findings

There were the following findings in relation to the controller firmware:

- The firmware must implement the USB transport protocol (see chapters 10.1 and 13) and the application protocol in Appendix D.
- In doing so, the firmware must report a specific set of USB descriptors so as to enable the host computer to load the correct drivers (see chapter 11).

4.3 Software findings

There were the following findings in relation to the host computer software:

- The device driver should be built on the generic “WinUSB” kernel-mode driver included with Windows (see chapter 11).
- The device driver must implement the application protocol in Appendix D.

4.4 Standards and law

The following standards and laws were identified as being relevant:

- The 4-Wire Pulse Width Modulation (PWM) Controlled Fans Specification, revision 1.3 (Intel Corporation, 2005a).

- SFF-8551J — Form Factor of 5.25” CD Drives, revision 3.3 (SFF, 2000).
- The Universal Serial Bus Specification, revision 2.0 (USB-IF, 2000).
- The Universal Serial Bus Device Class Specification for Device Firmware Upgrade, version 1.1 (USB-IF, 2004).
- The PCI Express® 225 W/300 W High Power Card Electromechanical Specification, revision 1.0 (PCI-SIG, 2008).
- The General Product Safety Regulations 2005 (S.I. 2005/1803).
- The Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment Regulations 2012 (S.I. 2012/3032).
- The Electromagnetic Compatibility Regulations 2016 (S.I. 2016/1091).

The General Product Safety Regulations 2005

Regulation 3 of the General Product Safety Regulations 2005 provides that every product is subject to the regulations “in so far as there are no specific provisions with the same objective in rules of [European Union] law” (unless it is covered by the exemption in regulation 4). While the definition of “product” in regulation 2 would exempt a prototype for a fan controller, any production design would fall within the definition, and so the regulations would apply to a production design.

Part 2 of the regulations sets out a general requirement for products to be safe, and more specific requirements for the producer to provide a customer with such information necessary for the customer to assess and take precautions against the risks inherent to the product, for the producer to take measures which enable the producer to be informed about the risks (and take such action necessary to avoid or mitigate the risks), and for the producer to inform an enforcement authority if the product “poses risks to the consumer that are incompatible with the general safety requirement.”

The Electrical Equipment (Safety) Regulations 2016

Although they were identified as potentially relevant, the Electrical Equipment (Safety) Regulations 2016 (S.I. 2016/1101) were determined to not apply to the fan controller—the regulations are, per regulation 3 (electrical equipment to which these regulations apply), applicable only to “electrical equipment designed for use with a voltage rating of between 50 and 1000 V for alternating current and between 75 and 1500 V for direct current.”

The Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment Regulations 2012

These regulations implement the European Union “RoHS directive” into United Kingdom law, and—as indicated by the title—restrict the use of certain materials and substances in electrical and electronic equipment. Under regulation 5 (EEE[1] to which these regulations apply), the restrictions apply only to equipment which is or was “placed on the market,” and so would not apply to the fan controller.

However, considering requirement 2.1.8 (which sets a target for cost in volume

[1] Abbreviation for “electrical and electronic equipment” used in the regulations.

production), it would make sense to ensure that any final prototype uses only parts which comply with the regulations and the RoHS directive.

The Electromagnetic Compatibility Regulations 2016

Under regulation 3 of the Electromagnetic Compatibility Regulations 2016 (but subject to regulations 3(2) to (4), 4, 5, and 6), the regulations are applicable to "all equipment." Fundamentally, the regulations require that equipment not generate "electromagnetic disturbance [...] [exceeding] the level above which radio and telecommunications equipment or other equipment cannot operate as intended," and require that equipment be designed so that it is able to "operate without unacceptable degradation of its intended use" when subjected to levels of electromagnetic disturbance "to be expected" in its intended use.

While such requirements would apply to any final design, it is possible that a prototype design would fall under one of the exemptions in the regulations. For example, regulation 3(e) exempts "custom built evaluation kits destined for professionals to be used solely at research and development facilities for such purposes," and regulation 5 exempts equipment on "display or demonstration at a trade fair, exhibition or similar event [...] provided that a visible sign clearly indicates that the equipment" does not comply with the requirements.

In light of this, and again considering requirement 2.1.8, any designs should have regard to the requirements for electromagnetic compatibility.

The Ecodesign for Energy-Related Products Regulations 2010

These regulations implement the "Ecodesign" directive of the European Union, require a device to conform to a number of European regulations, and require a device's declaration of conformity to state which of the regulations to which the device conforms (S.I. 2010/2617).

In particular, it was considered that a fan controller may fall within the definition of "electrical and electronic household and office equipment." However, per the definition in article 2 of Commission Regulation (EC) No 1275/2008 (which is referenced by the 2010 Regulations), a fan controller does not fall within that definition because it is neither one of the items in the list given in Annex I to the Commission Regulation, nor is it likely to be "dependent on [...] the mains power source in order to work as intended."

Despite it not appearing to be the case that a fan controller would be covered by the regulations, it would not be unreasonable to use the requirements given in that Commission Regulation—and any other listed by the 2010 Regulations which appears relevant—as targets. In any case, any design for a fan controller should have regard to the objectives of the Regulations.

5. Computer fans

5.1 Types

There exist few citable sources of information for the types of computer fan in common use. The types which exist are largely distinguished by the connector used. Including with the use of prior knowledge, the following types of fan were identified:

- 4-pin (or "PWM") fans
- 3-pin fans
- 2-pin fans
- ATX Peripheral Power Connector[2] (ATX-PPC) fans

Fans of the 4-pin type comply with an open specification (Intel Corporation, 2005a). Although no specification was found for the 2- and 3-pin types, there does exist documentation from computer motherboard vendors confirming that 3- and 4-pin types use a compatible connector (Intel Corporation, 2013). There also exists documentation implying, when taken with the aforecited, that the 2-pin connector type is compatible with 3- and 4-pin types (Intel Corporation, 2007).

ATX-PPC fans are powered using the Peripheral Power Connector described in the ATX Specification, although the use of the connector for fans is not known to be required or recommended by any specification.

ATX-PPC fans are sufficiently uncommon to not warrant specific support. In a search of a number of online retailers to assess commonness, few ATX-PPC fans were available, and those that were available were generally unbranded or of a generic brand. While the same is true of 2-pin fans, such fans can be connected to 3- or 4-pin headers, and so should be supported for completeness. ATX-PPC fans use an entirely different connector.

5.2 Construction

Manufacturers of computer fans provide little information about the construction of their fans, and very few provide manuals or datasheets. As such, precise information on the construction of a typical computer fan is not available and instead there must be assumptions and best guesses.

In determining the variety of motor used, several varieties can be disregarded from the outset:

- Any type of A.C. motor is highly unlikely. A typical computer power supply only provides a low-voltage D.C. output (Intel Corporation, 2002, p. 19), and so a fan with an A.C. motor would require an integrated inverter and, potentially, an integrated transformer[3] or boost converter.

[2] The ATX Peripheral Power Connector is known colloquially as the "Molex connector."

[3] For clarity, it is considered that an integrated transformer may be required as the highest D.C. voltage available (12V) is, if taken as the peak voltage of the A.C. waveform, would, after inversion, produce a very low A.C. voltage of approximately 8.5 V (RMS).

- Any brushed D.C. motor is unlikely, as the motor would require regular maintenance or replacement as its brushes wore down. In addition, the conductive carbon dust produced by the brushes wearing down could be ejected from the fan into the computer chassis, where it would be likely to land on exposed circuit boards and potentially interfere with their operation.
- Discounting the previous point, the use of a series-wound motor (including a universal motor) is unlikely given that the motor speed decreases with load. The light load of a fan might then cause a series-wound motor to accelerate past safe operating speed.

It is therefore reasonable to assume that most, if not all, computer fans make use of brushless D.C. motors (BLDCs). To attempt to confirm this assumption, a list of computer fans was obtained from an online retailer, and an attempt was made to locate the datasheets for each of the fans. The list is given in Table 1.

Table 1

Types of motor used in computer fans

Fan Model No.	Datasheet	Motor Type
`ADDA Corp. AD0912US-A70GL`	Yes (2004)	Brushless
`Arctic AFACO-080PC-GBA01`	No	—
`Corsair CO-9050059-WW`	No	—
`Noctua NF-A20 PWM`	Yes[4] (2017a)	Brushless
`AKASA AK-191-BL`	No	—
`Coolermaster R4-LUS-07AB-GP`	Yes (2008)	Not Specified

In summary, where a datasheet could be located and that datasheet specified the type of motor in use, no fan was specified as using any type other than “brushless” or an equivalent.

5.3 Control

For fans of the 4-pin type, the fourth wire is used to transmit a Pulse Width Modulation (PWM) control signal to the fan (Intel Corporation, 2005a, p. 9). Refer to chapter 6 for further information on PWM.

No control wire is provided with 2- or 3-pin type fans (Intel Corporation, 2007, p. 52; 2013). This lack of a control input could cause complication, as the speed of a brushless D.C. motor (which chapter 5.2 established as being the most likely choice of motor) is generally controlled by an on-board controller adjusting in sequence which poles are energised. An alternative method of speed control must be determined.

Speed control for 2- and 3-pin type fan motors should be possible by modulation of the supply voltage provided to the motor, as “the speed of the motor can be controlled if the voltage across the motor is changed.” Through the use of PWM, the average voltage across the motor can be reduced to give the same reduction

[4] Although not expressly stated, the “Smooth Commutation Drive” technology is listed in the datasheet and it, as marketing material explains, is for brushless motors (Noctua, 2017b).

in speed (Gamazo-Real, et al., 2010, pp. 6902, 6908-6909). Further, unlike if the absolute voltage were reduced, the use of PWM will not reduce the torque of the fan motor.[5]

However, as explained in chapter 5.4, the use of PWM is likely to be unsuitable for speed control of 3-pin type fans. The same effect must be accomplished by other means, such as reducing the absolute (rather than average) voltage supplied to the fan. Reducing the absolute voltage would not be without its downsides, however, as the reduced absolute voltage would also result in a reduced torque being produced by the motor. This effect is not present in a PWM-controlled motor as the full operating voltage is always supplied. As a result of the reduced torque, a 3-pin type fan would not necessarily support speed control in the 30% to 100% speed range supported by 4-pin type fans.

5.4 Monitoring

In order to both control a fan and report its status, it is necessary to be able to monitor its speed. This is especially true if precise speed control is desired, as fans of the 4-pin type are specified as having their speed correspond, plus or minus 10%, to the duty cycle of the PWM signal transmitted on the control wire (Intel Corporation, 2005a, p. 14).

For the 3- and 4-pin type fans, speed monitoring is enabled by a tachometer wire provided as the third pin. Given that 3-pin types are described as working when connected to a 4-pin header, it can be taken that both 3- and 4-pin types provide the same form of output on the tachometer wire. The tachometer wire for these types of fans provides two pulses for every revolution (Intel Corporation, 2005a, p. 9; 2013).

However, for 3-pin fan types where no speed control input is present, the use of PWM on the supply has the potential to interfere with the speed measurement device. Such devices are typically Hall effect sensors, which rely on a magnet exerting a force on the current in a current-carrying conductor such that a measurable voltage proportional to that current and the strength of the magnetic field is produced at a 90° angle to the direction of current flow (Gamazo-Real, et al., 2010, p. 6903; Honeywell, n.d., p. 3).

All located documentation regarding Hall effect sensors notes that the sensors require a constant current, and hence a regulated supply, to give an accurate output. Figure 1 below illustrates an expectation of the output of a Hall effect sensor when a switching supply is used.

[5] Briefly, the "on" periods of a PWM signal are at the full voltage. Thus, for the periods where the signal is "on," full power is delivered. See chapter 6 for further detail on PWM.

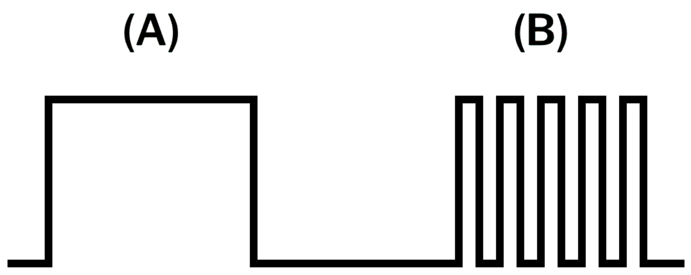


Figure 1
A comparison of the output of a Hall effect sensor when supplied with a constant current (A) and the expected output when supplied from a PWM-modulated supply (B)

If this is the case, and if a sufficiently high switching frequency is used, it may be possible to use some form of low-pass filter to remove the switching-frequency noise and recover the basic sensor output desired.

5.5 Power requirements

Fans of the 4-pin type are specified as requiring an input voltage of 12V, within 5%, and are permitted to draw up to 1.5 A during continuous operation. During starting, it is permitted for the fans to draw up to 2.2 A for one second (Intel Corporation, 2005a, p. 9). Given the compatibility of 2- and 3-pin types with 4-pin type fans, it can be taken that those types will largely adhere to the same requirements and limitations.

6. Pulse width modulation

Pulse width modulation (PWM) is a technique for approximating an analogue signal by altering the ratio between the time ("width") a digital signal is on and the time it is off. By altering that ratio, the average voltage can be reduced to a level between the digital system's "off" (generally 0 V) and "on" levels. The ratio is generally known as the "mark–space ratio," with each "on" level forming part of a "mark" and each "off" level forming part of a "space."

The mark–space ratio is commonly represented as the "duty cycle," expressed as the percentage of the period that the signal is "on." For example, to half the average voltage of the signal, the mark–space ratio would be 1:1 and the duty cycle 50%. For a three-quarters average voltage, the ratio would be 3:1 and the duty cycle 75%.

A PWM waveform is illustrated in Figure 2. The waveform has an 80% duty cycle, and—as can be seen in the figure—the voltage is at "V+" (or on) for three fifths of the period *T*, which is equivalent to 60%. It can also be seen that the voltage is at "V–" (or off) for two fifths of the period *T*, and by considering these two fractions together it can be seen from where the 3:2 mark–space ratio comes.

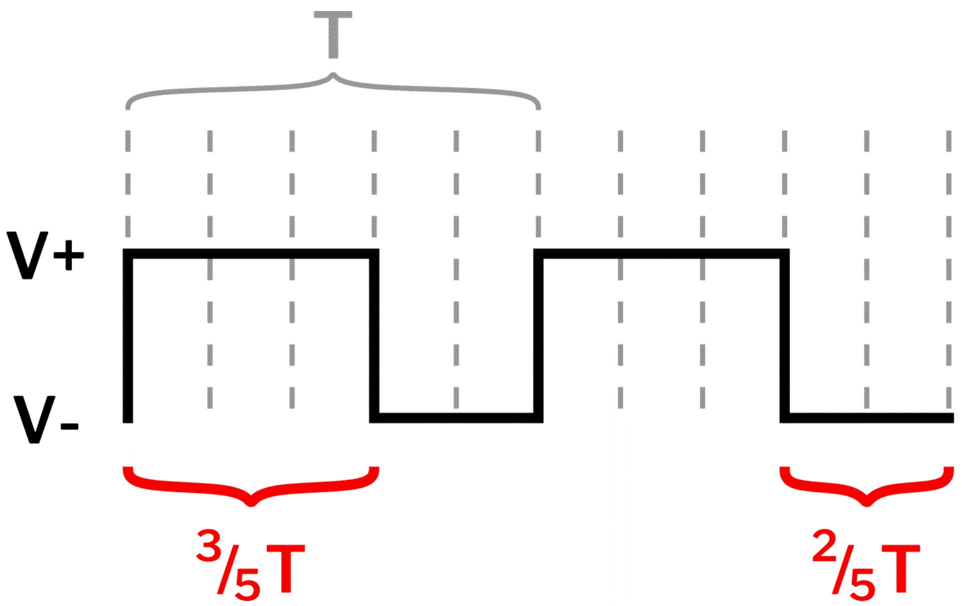


Figure 2
A PWM waveform at 60% duty cycle, 3:2 mark–space ratio

The PWM frequency must be chosen to ensure that the signal receiver state does not significantly deteriorate between pulses. For example, to use PWM to dim a light, the selected PWM frequency must be high enough that the human eye is unable to perceive the light switching on and off (Giesselmann, et al., 2002).

7. Form factor

In order to fulfil requirement 2.1.3, the controller must be suitable for mounting in a typical computer chassis. The following locations have been identified:

7.1 The rear expansion card slots, up to seven of which may be provided in a typical chassis (Intel Corporation, 2002, p. 11).

7.2 The 2.5” bays, often at the front but sometimes on the floor of the chassis.

7.3 The 3.5” bays, typically at the front of the chassis.

7.4 The 5.25” bays, typically at the front of the chassis.

7.1 Expansion card slots

The selection of location 7.1 would generally only be acceptable—in terms of what is and what is not good practice—if the host–controller interface selected were PCI or PCI Express. Those interfaces were rejected, and so location 7.1 must also be rejected. See chapter 9.1 for rationale.

Even if the PCI or PCI Express expansion card slots were selected as the mounting location, it is possible and likely that there would be other expansion cards present in the host computer. The number of cables running to the controller in a PCI or PCI Express slot—at least one for power, one for the host–controller interface, and four for the four fans—could cause considerable space issues and impact to airflow inside the host computer chassis.

7.2 2.5", 3.5", and 5.25" bays

A list of computer chassis was obtained from an online retailer, and a survey of the available quantities of 2.5", 3.5", and 5.25" bays was taken. The results are shown in Table 2.

Table 2

Availability of various sizes of bay in typical computer chassis

Chassis Model No.	No. of Bays		
	2.5"	3.5"	5.25"
Fractal Design FD-CA-CORE-3000-BL	2	3	2
Cooler Master SIL-652-KKN2*	3 + 7	2 + 7	3
Thermaltake CA-1H8-00M1WN-00*	3	3	0
Corsair CC-9011016-WW*	6	6	3
AvP CAS-748	2	4	2
Antec VSK4000B U3	0	5	3
Mean	4	5	2
Std. Deviation	3.6	2.3	1.2

* Has combination 2.5"/3.5" bays, which can be used in either configuration.

From the survey, it can be seen that the most common type of bay is the 3.5" bay. However, quantity is not the only factor which must be considered. Each type of bay is generally used with one specific component, and, although no source could be found for confirmation, prior knowledge and experience indicates:

- That 5.25" bays are typically used for optical media readers, and occasionally for memory card readers.
- That 3.5" bays are used for almost all hard disk drives.
- That 2.5" bays are used for a majority of solid-state drives, although some solid-state drives use motherboard-mounted connectors and some hard disk drives are available in the 2.5” form factor.

It can be reasoned that a user is likely to have fewer available 3.5" bays than 5.25" bays—digital media is increasingly delivered online and on-demand rather than by optical media, and so the demand (and therefore need) for optical media readers is decreasing (Morris, 2016; Sweney, 2017). It can be further reasoned that as digital delivery increases, so too will the need for storage space, and so the availability of 2.5" and 3.5" bays could decrease further as users add to the storage capacity of their computers.

This is especially important considering the deviation in the number of bays. In the survey, the chassis were consistent in the number of 5.25" bays provided. The surveyed chassis were relatively consistent in the number of 3.5" bays, but were not greatly consistent in the number of 2.5" bays provided. If the number of bays is not consistent, then the confidence with which it can be assumed that a user will have a bay of that type available is decreased.

Taking this into consideration, the controller must be of a form factor which can be mounted in a typical 5.25" bay.

The 5.25" device form factor was previously developed by the Small Form Factor Committee (SFF), now the Storage Networking Industry Association (SNIA)'s SFF Technology Affiliate Technical Working Group (SFF TA TWG). Various revisions of the SFF specification have been standardised in the Electronic Components Industry Association's EIA-741 standard, but this report will design to the latest version of the specification published by the SFF. For the greatest compatibility, the standard for 5.25" CD drives will be used (SFF, 2000).

8. Power delivery

8.1 Sources

It can be reasonably expected that any desktop- or tower-style computer will include a power supply. It can also be reasonably expected that any such power supply will, minimally, provide the power connectors referenced by the ATX specification,[6] as many computer motherboards use the form factor (as evidenced by all computer motherboards given in Table 5 using ATX or an ATX-derived form factor).

However, neither the ATX specification nor the ATX12V specification[7] (Intel Corporation, 2005b) provide any minimum power requirement. While this makes sense and is reasonable—many types of computer exist with vastly different power requirements—its effect is that all work of selecting an appropriate power supply is delegated to the user of the controller, who must then account for the power usage of the remainder of the computer system, the power usage of each fan connected to the controller, and the capabilities of each power rail provided by the user's power supply.

Use of standard ATX power connectors and supplies is, therefore, not ideal. The following power connectors were identified as providing a standard minimum amount of power:

- USB
- SATA
- PCI-E

USB power

As established in chapter 9.4, the USB interface of a computer (which is available in the internals of the computer) can provide power. However, USB 2.0 is limited to providing 5 V and 500 mA (2.5 W), which is neither sufficient voltage nor sufficient current to power a single fan. USB 3.1 can provide 5 V and 900 mA (4.5 W), but this is still not sufficient for a single fan. It could be possible to use

[6] ATX is the most common computer motherboard form factor. The ATX specification describes the required motherboard sizes, shapes, and mounting hole placements, among other requirements. All computer motherboards listed in Table 5 use the ATX (or an ATX-derived) form factor.

[7] The ATX12V specification provides supplementary information for power supplies compliant with the ATX specification.

supply from the USB connection providing the host–controller interface for the low-power subsystems on the controller.

SATA power

Serial ATA (SATA) is one, and probably the most prevalent, of the standards for connecting storage devices to computers. The SATA standard specifies two types of connector—data and power—for connecting storage devices, with two primary designs for each.[8] The SATA power connector is specified as being rated for up to 1.5 A per conductor, and provides three 12 V conductors, three 5 V conductors, and three 3.3 V conductors, and so would be able to power three computer fans directly from its 12 V supply (albeit with no margin if a fan were to draw the maximum permissible current).

To supply the fourth fan required by requirement 2.1.2, it would likely be possible to use a boost converter powered from one of the lower-voltage sets of conductors to produce an appropriate 12 V source (SATA-IO, 2009, pp. 67, 82, 95).

However, two issues were identified with the use of the SATA power connector. First, for sufficient margin on the power supplied to the computer fans, it would be necessary to either have two power connectors or a high-rated boost converter which could provide sufficient power for a fourth fan and the overhead for all four fans; and secondly, no manufacturer could be found which produced a power-only receptacle, and so each connector on the controller would include a non-functional SATA data connector.

PCI-E power

Although PCI-E was rejected as a host–controller interface (see chapter 9.1), the power connectors specified to be used with PCI-E devices remain an option for sourcing power, as there is no requirement that a device powered from a PCI-E power connector be a PCI-E device.

The PCI-E specification defines two auxiliary power connectors: the "2 × 3" connector (up to 75 W) and the "2 × 4" connector (up to 150 W), both of which provide this power at 12 V (PCI-SIG, 2008). It would be possible to use either connector to power the fans connected to the controller: if the 2 × 3 connector were used, the USB power available via the host–controller interface connection could be used to power other components on the controller, and if the 2 × 4 connector were used the power it supplies could be used to power the entirety of the controller's components. A survey of typical power supplies sold by an online retailer was taken to determine the availability of the types of auxiliary power connectors (Table 3).

[8] The remaining designs are generally for smaller storage devices, such as those using the 1.8" form factor, or for storage devices external to the host computer (SATA-IO, 2009, pp. 111, 120, 138, 147).

Table 3
Availability of PCI-E auxiliary power connectors on typical computer power supplies

Power Supply Model No.	**Rating**	**No. of Connectors**	
		2 × 3	**2 × 4**
`Corsair CP-9020095-UK*`	350 W	0	1
`Antec VP350P`	350 W	1	0
`be quiet! BN240*`	400 W	0	2
`Seasonic SS-430ST*`	430 W	0	1
`Corsair CP-9020101-UK*`	450 W	0	1
`EVGA 100-W1-0500-K3`	500 W	1	0
`be quiet! BN277*`	500 W	0	2
`EVGA 220-G3-0550-Y3*`	550 W	0	3
`Corsair CP-9020076-UK*`	550 W	0	2
`Antec EDG550*`	550 W	0	2
`be quiet! BN278*`	600 W	0	4
`Corsair CP-9020122-UK*`	650 W	0	2
`EVGA 210-GQ-0750-V3*`	750 W	0	6
`be quiet! BN237*`	800 W	0	4
`Seasonic SSR-850TD*`	850 W	0	6
`Corsair CP-9020037-UK*`	860 W	0	8
`EVGA 220-G3-1000-X3*`	1000 W	0	8
`be quiet! BN254*`	1000 W	1	8
`Corsair CP-9020008-UK*`	1200 W	0	8
`Corsair CP-9020057-UK*`	1500 W	0	10

* Uses combination "6+2-pin" PCI-E auxiliary power connectors.

It can be seen that, as rated power increases, so too does the number of available PCI-E auxiliary power connectors. Especially outside of the relatively low end of power ratings (300–450 W), most power supplies have multiple available power connectors. It would therefore be reasonable to assume that most systems will have at least one additional PCI-E power connector available, especially at the higher end as PCI-E add-in cards rated at 225–300 W are specified as being able to use no more than two auxiliary power connectors (PCI-SIG, 2008, p. 16).

For the best compatibility, the controller would use the 2 × 3 auxiliary power connector. While this would prevent use with any power supply which provides only 2 × 4 power connectors (PCI-SIG, 2008, p. 22), all power supplies surveyed which provided a 2 × 4 connector provided a combination "6+2-pin" connector.

The 6+2-pin connector is a modified variant of the 2 × 4 connector which has a removable 2-pin segment. When the segment is present, the connector can be inserted into a 2 × 4 receptacle. When the segment is removed, the connector becomes compatible with the receptacle for the 2 × 3 connector. Unfortunately, as the 6+2-pin connector is a *de facto* standard, there could not be identified a suitable source to cite to confirm this compatibility.

Selection

The controller, for the reasons outlined, will use a PCI-E 2 × 3 auxiliary power connector as the 12 V source to power the attached fans, and the USB connection that is the host–controller interface for a 5 V source to supply the lower-power components.

8.2 Fans

Power supply margins

A compliant fan is permitted to draw 1.5 A continuously and up to 2.2 A during starting. Requirement 2.1.2 specifies that the controller must support four fans, and this would appear to leave little safety margin—four fans would be permitted to draw 6 A continuously and 8.8 A if all started at once. However, in reality, a modern fan will draw considerably less than the maximum permitted. A survey of fans sold by an online retailer was taken to confirm this (Table 4).

Table 4

Continuous-operation current draw for typical computer fans

Fan Model No.	Speed	Current
`Cooler Master R4-LUS-07AR-GP`	700 rpm	0.160 A
`Aerocool 4713105951615`	1000 rpm	0.200 A
`Bitfenix BFF-SCF-12025KK-RP`	1000 rpm	0.100 A
`Fractal Design FD-FAN-DYN-GP12-WT`	1200 rpm	0.180 A
`Thermaltake CL-F038-PL12RE-A`	1500 rpm	0.200 A
`Noctua NF-A14 PWM`	1500 rpm	0.130 A
`Noctua NF-F12 PWM`	1500 rpm	0.050 A
`Noctua NF-B9 REDUX PWM`	1600 rpm	0.080 A
`Phanteks PH-F120MP`	1800 rpm	0.200 A
`be quiet! BL051`	2000 rpm	0.100 A
`Cooler Master R4-JFDP-20PW-R1`	2000 rpm	0.400 A
`Corsair CO-9050013-WW`	2350 rpm	0.180 A
`EKWB 3831109880036`	3000 rpm	0.470 A
`Noctua NF-A4x10 FLX`	4500 rpm	0.050 A
	Mean	0.179 A
	Std. Deviation	0.122 A
	Max.	0.470 A
	Mean Amps/1000 rpm	0.113 A

As shown by this data, it is unlikely that a modern fan operating normally will come close to the maximum permitted current draw. Even the mean plus three standard deviations (0.545 A) is well within the capability of the power supply and still provides a large margin of safety.

Further, assuming the mean current per 1000 rpm figure holds true, there would need to be connected a fan that rotated at approximately 13,300 rpm. While such fans do exist, the current per 1000 rpm figure does not hold true for those fans and use of those fans would still provide an acceptable margin of safety. For example:

- The NMB-MAT 1611RL-04W-B60-X00 (15,000 rpm) draws 0.36 A.
- The Delta THA0412 (19,000 rpm) draws 0.89 A.
- The Sanyo Denki 9GX3612P3K001 (24,000 rpm) draws 1.30 A.

However, these fans are not intended for use as computer fans, as evidenced by the lack of the standard computer fan connector and their control wires having differing functions; and as further evidenced by another Sanyo Denki fan in the same series far exceeding the permitted maximum current draw for computer fans—the Sanyo Denki 9HVA0812P1G001 (16,100 rpm) is rated to draw 3.5 A.

Speed control by power delivery

As discussed in chapter 5.3, fans of the 2- and 3-pin types do not provide a wire for control, and so the power supply voltage delivered to the fan must be reduced to reduce the speed. However, as noted in chapter 5.4, simply using the a PWM signal (see chapter 6) to switch on the power supply wire could interfere with instrumentation on the fan leading to the reporting of an inaccurate speed.

Potential methods for filtering the PWM output to produce a smooth waveform were investigated, and all potential methods investigated were based on the same concept—that a PWM signal would be filtered to produce an absolute voltage approximately equal to its average voltage—with the main difference being in how this was used to deliver power.

A so-called "PWM DAC" (pulse width modulation digital-to-analogue converter) is a circuit generally found in systems where an integrated DAC is unavailable and high-resolution output is not required. Given a PWM generator, a PWM DAC can be implemented with relatively inexpensive components, such as a resistor and capacitor forming a low-pass RC filter.

The first potential method involved using the output of the PWM DAC to drive a bipolar junction transistor (BJT) in the common collector configuration. A BJT in this configuration acts as a voltage buffer, with voltage gain near unity. The emitter voltage of the BJT is equal to the base voltage (i.e. the DAC output) minus the forward voltage (generally 0.6–0.7 V for silicon transistors). However, this method results in high losses in the transistor, with the power dissipated in the transistor P_D being $I_C \times V_{CE}$—that is, the product of the collector current and the voltage between the collector and emitter. Hence, at 6 V base voltage and full operating current for a fan (1.5 A), the transistor could dissipate 9 W of power as heat (Horowitz & Hill, 1989, p. 65–69). During starting, before any filter in the PWM DAC has had sufficient time to stabilise, the base voltage may be near-zero, meaning that nearly the full supply voltage is dropped on the transistor. This, coupled with the fan starting current of 2.2 A, means that the transistor could potentially dissipate more than 26 W, plus any further power loss from the base current. This method is therefore not viable.

The second potential method did away with a transistor acting as a voltage buffer and instead used the PWM DAC directly to drive the fan. If an RC (resistor–capacitor) filter were used, the resistance could be lowered and the capacitance increased to ensure that the cut-off frequency remained the same. The power dissipation would then be only the ohmic heating in the resistor and capacitor $P=I^2R$. While capacitors rated in the millifarad range—the anticipated range for this method—are relatively costly, a low aggregate resistance on the order of 400 milliohms would then result in a worst-case power dissipation (at the starting current 2.2 A) of 2 W. The downside to this method is that the capacitor charging current would be only limited by that aggregate resistance. At 400 milliohms, this could result in 30 A being drawn—far above the level the power supply is rated to supply. While there do exist components to limit the inrush current, they are not inexpensive—one solution from Texas Instruments (the TPS2420) cost more than £2 in single quantities.

There are other methods of mitigating inrush current—for example, by adding an inductor to make an LC filter, not only can the capacitance in the filter be reduced (which will reduce inrush current), but the inductor's opposition to changes in current will damp any spike in current demand.

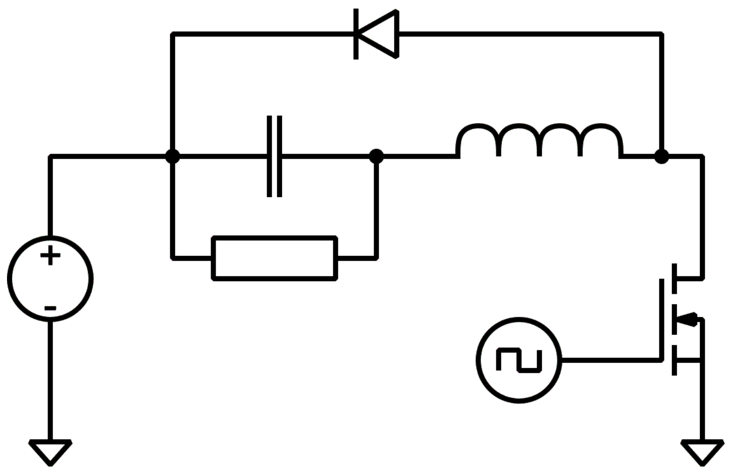


Figure 3

A generic MOSFET-controlled PWM DAC based on an LC filter

However, simulation of such a circuit revealed that, if the PWM DAC were operated at 100% duty cycle, the voltage across a load (in the simulation, represented by a 4-ohm resistor) would spike above the rated maximum (12.6 V) of a fan to 16 V during starting. Whether this method is viable, then, depends on whether this voltage can be snubbed so that it remains within acceptable bounds.

The circuit shown in Figure 3 was simulated, and, in that simulation, increasing the inductance in the LC filter reduced the voltage spike considerably—from 16.8 V (at 1 mH) to 12.2 V (at 8 mH). At 10 mH, there was no discernible spike and the voltage across the load was within millivolts of the 12 V target. There is likely to be some variation with real components, but it is expected that any variation would be downwards.

In this circuit, the diode is necessary to prevent damage to the circuit—if it were not present and the MOSFET was switched off, the inductor would resist the sudden change in current and continually increase its voltage in an attempt to retain an unbroken circuit. This high voltage could damage the transistor and other components, but with the diode in place a path is provided for current to circulate, and the inductor will be continually charging and discharging until its energy is dissipated.

Discussion on the selection of components for the PWM DAC can be found in chapter 10.2.

8.3 Microcontroller

The power delivery system for any microcontroller is uncomplicated. The USB connection that is the host–controller interface (see chapter 9) can provide up to 500 mA at 5 V, which is likely to be sufficient for any microcontroller that could be selected.

Where a microcontroller cannot accept a 5 V supply voltage, the level of power in the circuit is slight enough that a linear regulator—that is, a regulator which uses a varying resistance to drop the voltage required to produce the desired output (and so is heated by resistive heating per Joule's law $P=I^2R$)—is acceptable as a solution.

Available current may be somewhat limited if more than the controller and its ancillary components are to be supplied from the USB 5 V supply. However, it is not anticipated that this will be the case.

9. Host–controller interface

In order to be easily programmable, the controller must interface with the host computer and must be able to receive instructions from that computer. There are numerous interfaces in use in a typical computer system, and the following have been identified as potential choices:

- PCI or PCI Express
- RS-232
- Ethernet
- USB

From these potential choices, USB has been identified as the most appropriate. The following sections provide the rationale for this decision.

9.1 PCI and PCI Express

The PCI and PCI Express (PCI-E) buses would, on first consideration, appear ideal interfaces for the controller. They provide sufficient power (up to 300W), a standardised slot in which the controller could be mounted, and a high-speed communications interface to the host computer.

However, modern computer motherboards seldom include PCI slots. When PCI slots are included, generally few are included, reducing the likelihood that one slot might be free. In a survey of computer motherboards sold by an online retailer (Table 5), PCI slots were so scarce as to be effectively unavailable.

Table 5

PCI and PCI Express availability on computer motherboards

Motherboard Model No.	No. of Slots	
	PCI	PCI-E
Gigabyte GA-H110-D3A	0	6
MSI Z270-A-PRO	0	6
Asus 90MB0PB0-M0EAY0	2	4
Asus PRIME X370-PRO	0	7
ASRock AB350M-HDV	0	2
Gigabyte EBR1-GA-AB350-GAMING 3	0	5
Biostar X370GT5	2	4

However, as can be seen, most of the computer motherboards surveyed included a number of PCI-E slots. In terms of availability, then, PCI-E is a viable option. The same is not true of PCI-E when the technical requirements are taken into consideration. The bus uses a 100 MHz reference clock, which is relatively high frequency in and of itself, and transfers data on 2.5GHz, 5GHz, or 8GHz channels (PCI-SIG, 2010, pp. 354, 401). It is unlikely that any microcontroller appropriate for the controller would support the bus, and, even if a microcontroller with support for the bus were available, specialist design knowledge and equipment would be required to implement and test the bus.

Disregarding the technical limitations, producing a PCI or PCI-E device requires the assignment of a Vendor Identifier (Vendor ID) by the PCI Special Interest Group (PCI-SIG), and to be assigned a Vendor ID a person or organisation must join PCI-SIG at a cost of $4000 per year (PCI-SIG, 2010, p. 588; 2017). Use of PCI or PCI-E is therefore impractical financially as well as technically.

9.2 RS-232

The use of RS-232 would be ideal for the controller—the interface is simple, operates at low speeds, and can be implemented either using the extremely common universal asynchronous receiver–transmitter (UART) hardware of a microcontroller or, if no UART hardware is available, with software running on the microcontroller and "bit-banging" (i.e. reading and setting the microcontroller input and output states, using software, in the way a dedicated transceiver would).

However, presence of connectors suitable for RS-232 on modern computers is limited, as the interface is largely obsolete outside of embedded and industrial applications. Using the same computer motherboards identified in Table 5, the availability of a suitable connector for RS-232 was assessed (Table 6).[9]

Table 6

Availability of RS-232 connectors on computer motherboards

Motherboard Model No.	**No. of RS-232 Ports**	
	DB-25	**DE-9**
Gigabyte GA-H110-D3A	0	1
MSI Z270-A-PRO*	0	0
Asus 90MB0PB0-M0EAY0	0	0
Asus PRIME X370-PRO	0	0
ASRock AB350M-HDV	0	0
Gigabyte EBR1-GA-AB350-GAMING 3	0	0
Biostar X370GT5*	0	0

* Includes a non-standard serial connector.

As can be seen, connectors suitable for use with RS-232 are effectively extinct in modern computer motherboards. In the one case where a standard connector was included, that connector was in the "back panel I/O" area and so would have been external to the host computer, requiring a cable leading from inside the host computer—where the controller is to be mounted—to outside of the computer, where the connector would be located.

RS-232 is therefore not a suitable choice of host–controller interface.

9.3 Ethernet

The use of Ethernet for control applications is common, and numerous industrial applications and specifications for Ethernet exist—EtherNet/IP, PROFINET, and Sercos III, for example, each use Ethernet and work with a typical network stack as would be found on a consumer-grade computer (Lin & Pearson, 2013).

No technical limitations preventing the use of Ethernet have been identified. The Ethernet specification allows for a wide range of signalling rates—from 16 MHz to 1000 MHz depending on the precise variant and data rate—and uses standard connectors and cable constructions (IEEE, 2015, pp. 77–78, 82, 401–402). Indeed, microcontrollers with integrated Ethernet hardware exist and are commercially available (such as the Microchip/Atmel AT32UC3A). There may be time-related issues if new software must be written to provide support for the Internet Protocol (which is used on most networks), but given the ubiquity of Ethernet and the Internet Protocol it is expected that such software already exists.

The limitations preventing the use of Ethernet are practical. An industrial computer is unlikely to be used for the same purposes as a consumer computer,

[9] A suitable connector is one of the either the DB-25 or DE-9 connectors. The EIA RS-232 standard specified up to 25 circuits for communication and 13 interfaces, where some of the circuits were omitted. It is also permitted for additional connections to be defined by mutual agreement (EIA, 1969, pp. 7–9, 21–22). The DE-9 connector was used by the IBM PC AT, but did not use any circuit configuration ("interface") defined in EIA RS-232.

and so there is no scarcity of Ethernet connections on an industrial computer. The user of a consumer computer, however, likely requires (or desires) use of the Internet, and so would be required to either have two available Ethernet ports (one for a connection to a router or network switch, one for a connection to the controller), or would be required to connect the controller (mounted inside the host computer) to a router or network switch.

Making further use of the computer motherboards from Table 5, a survey of the number of Ethernet ports available on a typical computer motherboard was made (Table 7). The survey demonstrated that most motherboards have only a single Ethernet port, indicating the first case is impractical.

Table 7

Availability of Ethernet ports on computer motherboards

Motherboard Model No.	No. of Ethernet ports
Gigabyte GA-H110-D3A	1
MSI Z270-A-PRO	1
Asus 90MB0PB0-M0EAY0	1
Asus PRIME X370-PRO	1
ASRock AB350M-HDV	1
Gigabyte EBR1-GA-AB350-GAMING 3	1
Biostar X370GT5	1

The second case—connecting the controller to a router or network switch—is undesirable for other reasons. Doing so could enable, intentionally or otherwise, communication with the controller over the Internet, and so the controller would be required to implement authentication to prevent a malicious actor from making remote changes to the configuration of the controller or exploiting any errors or vulnerabilities in the controller's firmware. This would not only impact the experience of a user, but would result in significantly more work in producing the controller.

Therefore, Ethernet is not a suitable host–controller interface.

9.4 USB

Universal Serial Bus (USB), as an interface, has a combination of the advantages of the previously-considered interfaces—it has low-speed operating modes that do not require specialist knowledge or tools, it is ubiquitous (Table 8), it has no dedicated use (e.g. as a network connection) which could bring unintended side effects, and it can provide power (USB-IF, 2000, pp. 1, 17, 171; 2013, pp. 9-9, 11-1).

Table 8

Availability of USB ports on computer motherboards

Motherboard Model No.	No. of USB Ports		
	USB 1.1 & 2.0	USB 3.1 (Gen. 1)	USB 3.1 (Gen. 2)
Gigabyte GA-H110-D3A	2 + 4	2 + 2	0 + 0
MSI Z270-A-PRO	2 + 4	4 + 4	0 + 0
Asus 90MB0PB0-M0EAY0	2 + 4	4 + 2	2 + 0
Asus PRIME X370-PRO	0 + 4	4 + 5	2 + 0
ASRock AB350M-HDV	2 + 2	4 + 1	0 + 0
Gigabyte EBR1-GA-AB350-GAMING 3	1 + 4	0 + 2	2 + 0
Biostar X370GT5	0 + 2	4 + 2	2 + 0
Mean	1 + 3	3 + 3	1 + 0

Note: Ports are given as external ports + internal headers. External ports are located on the back I/O panel of the computer and are standard USB ports, while internal headers are located on the motherboard are generally do not use standard USB ports.

The information in Table 8, taken with the backwards compatibility of USB 3.1 with USB 2.0, allows a USB device adhering to the USB 2.0 specification to be used in practically any computer with USB capability. The use of USB 2.0 is well within capability—such devices inherit the USB 1.0/1.1 “Low Speed” (1.5 Mbps) and “Full Speed” (12 Mbps) modes, which are sufficiently low speed that “bit-banging” is possible, if likely not advisable.

Further, USB has the advantage of “device classes,” which are standardised and generic specifications for types (or classes) of device. Any software designed to operate with a device class will, provided the device is compliant, be capable of operating any device of that class. In particular, the “Device Firmware Upgrade” (DFU) class provides a standardised method of updating the firmware on a USB device over a USB connection (USB-IF, 2004). This has the following advantages:

- In fulfilling requirements 2.2.4 and 2.3.3, the standardised nature of DFU means that there likely already exists firmware and software which could be incorporated or used as a reference, potentially reducing development time.
- A user could, if that user so desired, use software other than that produced for the controller to update the controller’s firmware.

USB also, and unlike RS-232 and Ethernet, has a standardised header for internal connections (Intel Corporation, 2005c, p. 27–29), and so there would be no need for external cables to connect the controller to the host computer.

There is a similar disadvantage with using USB as there is with using PCI or PCI-E, in that a USB devices requires a Vendor ID to function, and a Vendor ID can, ostensibly, only be obtained by paying a $4000 annual membership fee to the USB Implementers’ Forum, or by making a one-time payment of $5000 for the Vendor ID without membership (USB-IF, 2000, pp. 261-263; 2013, pp. 9-35–9-38; n.d.). However, this is not an issue in reality—numerous USB-IF members sublicense the Vendor ID they are assigned, and so a Vendor ID and Product ID could thereby be obtained for free. The following sublicensors were identified:

9.4.1 John Otander (2015)

9.4.2 OpenMoko, Inc. (2017)

9.4.3 FTDI Ltd. (2010)

9.4.4 Silicon Labs (2017c)

9.4.5 Microchip (2017)

9.4.6 NXP (2017a)

9.4.7 Texas Instruments (n.d.)

Each of the sublicensors imposes restrictions on who may apply to sublicense their Vendor ID. Sublicensors 9.4.1 and 9.4.2 generally require that the applicant apply only in relation to open-source projects, while sublicensors 9.4.3 to 9.4.7 require the use of their products, or license the Vendor ID up to a maximum number of produced units. It is not foreseen that there will be any difficulty in meeting requirements relating to open-sourcing the project.

For these reasons, USB is selected as the host–controller interface.

10. Preliminary hardware selection

The choice of components for use in the construction of the fan controller is a concern for the design specification, but for the primary system components it is necessary to provide the rationale behind the selection of that component. This chapter provides that rationale.

10.1 Microcontroller

The choice of controller affects the design of numerous parts of the system—the design of the firmware components (including the programming language with which the firmware is written) could change radically; the voltage tolerances of the controller could change whether a regulator is required for power, and if a level shifter is required for all fan interfacing; and the buses and interfaces the controller supports could change the choice of sensors and other devices.

In selecting an appropriate controller, there are the following primary criteria:

- The controller should support USB 2.0, whether at Low Speed (1.5 Mbps) or Full Speed (12 Mbps).
- The controller should have at least four PWM generators, although if timers are available it would be possible to use software to produce a PWM output. If PWM generators are present, they should ideally support multiple channels—that is, support adjusting the output pin through which the PWM signal is transmitted—and a variable PWM frequency.
- The controller should be inexpensive, and should have an inexpensive development kit to enable firmware and software to be written before any hardware design is sent for manufacturing.

- The footprint of the controller should be practical. While integrated circuits in extremely small footprints are available, larger footprints are often more practical—a larger spacing between the controller pins reduces the effect an error in manufacturing a circuit board could have, and requires less expensive capabilities from the fabricator; and small footprints with pins underneath the controller (such as BGA[10]) can require specialist examination techniques (including x-ray imaging) to verify correct assembly.

- Considering the limit of 0.5 A supplied from the USB connection, the controller should be of a low-power variety.

In the interests of saving cost and time, the controller selected is a controller from the Silicon Labs EFM32 "Wonder Gecko" family. While perhaps not the most economical option, controllers from the EFM32WG family fulfil the criteria set out above, are inexpensive, and a development kit for the family was already to hand. In particular, all devices in the family provide four timers (with no less than 11 PWM outputs), three pulse counters, at least three USARTs[11] which support the SPI bus, two I^2C buses, at least 32 kB of RAM, and at least 64 kB of non-volatile flash storage. All devices support five power modes, designated EM0 (full functionality, current draw from 10.8 mA) through EM4 ("shutoff mode" with all but interrupts and resets disabled, 20 nA minimum current draw), which fulfils the low-power criterion.

Many parts in the family provide USB 2.0 support, and those that do support both Low and Full Speed varieties. Parts with USB support also include an integrated 5 V-to-3.3 V regulator which can provide up to 50 mA (Silicon Labs, 2014a, p. 241).

The controllers use an ARM Cortex-M4F microcontroller core.

The development kit to hand—the Silicon Labs EFM32WG-STK3800—includes a higher-end device in the EFM32WG family. This mitigates the risk of, part way through development, discovering that a selected lower-end device is not suitable for this application. Development can take place on a higher-end device and, once all requirements are known with certainty, a compatible lower-end device can be selected for volume production. In producing a prototype, it is largely immaterial whether a higher-end device or the selected lower-end device is used (Silicon Labs, 2013d, p. 2; 2014a, pp. 8–11, 240–243).

10.2 PWM DAC

As established in chapter 8.2, each computer fan is permitted to continuously draw 1.5 A with peaks of up to 2.2 A with a supply voltage not exceeding 12.6 V. Any components in the PWM DAC, then, must be rated—at the absolute minimum—to withstand at least these currents and voltages. For safety, any components should be rated to tolerate currents and voltages at least 50% greater than these minimums.

As further established in that chapter, it is possible for the voltage across the load to spike on starting, and so the inductance must be chosen to ensure that any

[10] Ball Grid Array (BGA) — an arrangement of pins on an integrated circuit (or a "land pattern") where the die is mounted on a small circuit board and that circuit board has a grid arrangement of balls of solder on its underside (IPC, 1999, § 14).

[11] Universal Synchronous/Asynchronous Receiver/Transmitter (USART).

spike remains within the rated limits for the components, if it cannot be entirely eliminated. Experimentation by simulation revealed that by increasing the inductance (in this case, from 1 to 10 mH) for a fixed load, the spike was reduced such that it was largely indiscernible.

There are a number of components which must be selected for use in the PWM DAC—a capacitor, an inductor, a flyback diode, and a transistor. If desired, it may also be necessary to select the relevant transducers for monitoring the output of the DAC for both control—ensuring the correct voltage is across the load—and for safety—ensuring that the current draw does not exceed the safe level for the power supply in use.

General filter considerations and selection

The selection of the capacitor and inductor is dependent on the desired characteristics of the filter. In particular, they must be selected to ensure that the cut-off frequency of the filter is such that the signal is sufficiently conditioned and that high-frequency noise does not impact the circuit's operation. At the very least, the cut-off frequency must be lower than the first harmonic—the PWM frequency—or there will be significant ripple on the output (Nisarga, 2011, p. 9).

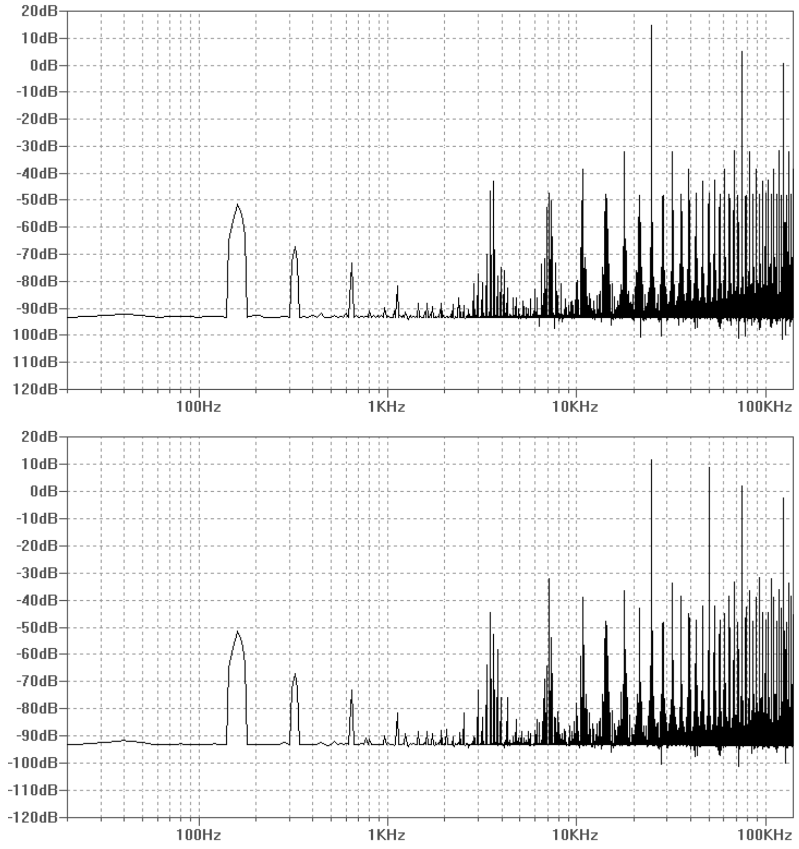


Figure 4
Fourier analysis of a 100 kHz PWM signal at 50% (top) and 75% (bottom) duty cycles

To better understand the signal and to aid in selecting a cut-off frequency for the filter, the simulation tool was used to perform Fourier analysis on a 100 kHz

PWM signal at 50% and 75% duty cycles. The result of the analysis is shown in Figure 4 and, as can be seen, the most significant frequencies begin slightly above 3 kHz. The cut-off frequency for the filter is therefore selected as 3 kHz. Using equations provided by a Texas Instruments report (2016, p. 9), the capacitance and inductance for the LC filter were calculated as in Figure 5. The resistance used in these calculations was determined as described in footnote 12.

$$\begin{aligned} L &= (R\sqrt{2}) / 2\pi f \\ &= (103.725\sqrt{2}) / (3000\times 2\pi) \\ &= 0.007782\ldots \\ &= 0.0078\text{ H} \\ L &= 7.8\text{ mH} \end{aligned} \qquad \begin{aligned} C &= 1 / (2\pi f R\sqrt{2}) \\ &= 1 / (2\pi\times 3000\times 103.725\times\sqrt{2}) \\ &= 0.00000036166\ldots \\ &= 0.000000362\text{ F} \\ C &= 362\text{ nF} \end{aligned}$$

Figure 5

Calculation of capacitance and inductance for LC filter

Unfortunately, this calculated inductance is entirely impractical—an inductor of inductance 7.8 mH would be impractically large and impractically expensive, and so cannot be used in the fan controller. However, it is anticipated that the effect will remain largely the same—the majority of any variation likely being as a result of the change in time constant and inherent or equivalent resistance—provided that any capacitance is increased in proportion with any decrease in inductance. Inspection of the stock list of an online retailer revealed the greatest selection of inductors below the 500 μH range, and so to maintain a comparable effect there must be selected a capacitance of $(7.8\times10^{-3}/500\times10^{-6})(362\times10^{-9}) = 5.65$ μF or more. However, given the typically-available values of capacitors, a 4.7 μF value is to be used, and so the inductance is to be $(362\times10^{-9}/4.7\times10^{-6})(7.8\times10^{-3}) = 600$ μH. The subsequent simulation of a circuit with these selected values produced a result comparable to simulation with the values from Figure 5, albeit with reduced damping. The further simulation with an inductance of 470 μH did not produce substantially different results, and so that value—found to be more common in the stock of the online retailer—can be used instead.

This modelling is imperfect—a computer fan is unlikely to exactly resemble a resistor. However, given that power is delivered to the fan motor through drive electronics, it is expected that modelling the fan as a resistance is reasonable. It will be necessary to confirm this through practical testing.

Further, while the selected inductances and capacitances are acceptable for loads equivalent to or equivalent to less than the 103.725-ohm weighted midpoint (as with such loads the filter will be overdamped or near-critically damped), it is not entirely suitable for loads equivalent to more than the weighted midpoint. For example, were a fan drawing 0.05 A to be connected (assumed equivalent to

[12] The load was modelled as a variable resistance for simplicity. The power required to be supplied to a fan was calculated using the fan current draw figures listed in Table 4 together with the 12 V nominal supply voltage. The mean starting voltages for those fans—calculated as 5.2 V—was treated as a worst-case voltage. An equivalent resistance was then found through $R=v^2/P$ at nominal voltage (12 V) for the lowest-current fan, and at worst-case voltage (5.2 V) for the current three standard deviations from the mean. These values are the highest- and lowest-likely equivalent resistances—240 and 4.135 ohms respectively. To find a single figure weighted based on the current draw of all identified fans, the mean of the two aforementioned resistances and the equivalent resistance for the mean current identified in Table 4 (67.04 ohms at 12 V) was taken. The resultant figure is 103.725 ohms.

$12^2 \times 0.05 = 240$ ohms), the filter would be severely underdamped—indeed, the Q factor for the filter in this case would be around 24 compared to the target of around 0.707. In such cases, as confirmed by simulation, the voltage supplied to the fan could greatly exceed the rated 12.6 V maximum. It is therefore necessary to include a snubbing circuit to limit any voltage spikes to within tolerance.

Snubber selection

A number of techniques for snubbing spikes in voltage exist but, for simplicity, the choice for this application is one of a Zener diode or transient-voltage-suppression (TVS) diode. The two are similar—both are able to conduct in the reverse direction without the damage a typical diode would sustain. There is also the advantage that use of these devices instead of other snubbers, such as those based around capacitors or an RC network, means that little consideration need be given to the effect on the LC filter the output of which the device is to snub.

As for the selection of a device, there are only a few factors to consider—the breakdown voltage (in the case of Zener diodes, more commonly called the Zener voltage V_Z), the power rating, and the maximum current rating.

The breakdown voltage rating is simple to determine. Computer fans are rated for up to 12.6 V with a nominal voltage of 12 V. Therefore, a breakdown voltage anywhere between 12 V and 12.6 V is acceptable. However, given that the power supply is specified as supplying up to 12.6 V (PCI-SIG, 2008, p. 14), it would be preferable to select a device with a breakdown voltage closer to 12.6 V so that the diode is less likely to continuously be in conduction.

The power and maximum current ratings are less simple to determine, and use of simulation is required so that an approximate figure for the voltage across the diode can be known. As a worst-case test, a 400-ohm load was placed in the test circuit (shown in Figure 3 with the capacitance and inductance from Figure 5) and simulated. This an approximate 66% margin over the 240-ohm maximum given in footnote 12 on page 32. The filter voltage is plotted in Figure 6, and the code used to make the plot is given in Appendix C1.

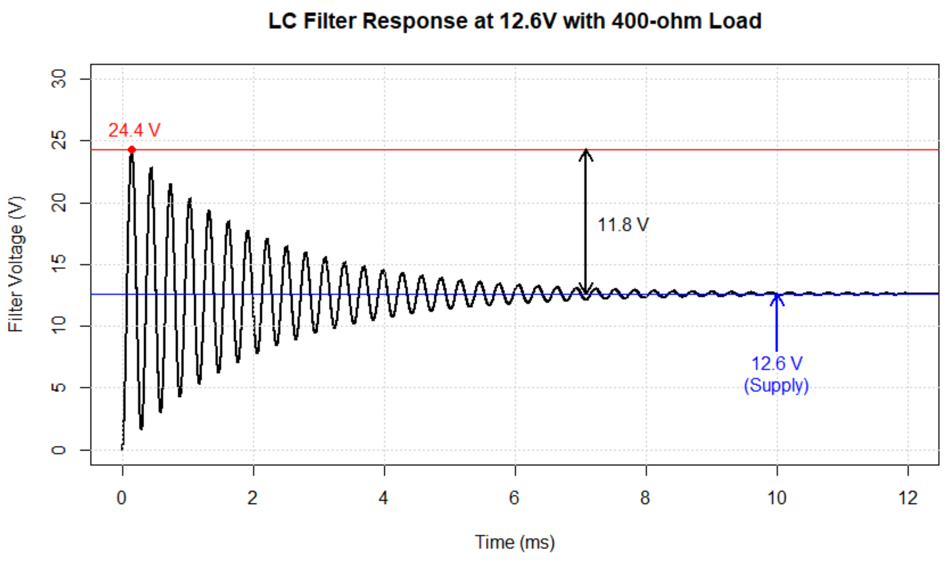


Figure 6

LC filter response at 12.6 V with a 400-ohm load

It can be seen that around 12 V could be placed across the snubber diode. While this is only for a fraction of a millisecond, there is the potential for a significant current to be produced, and a significant current could result in the power and current ratings of the diode being exceeded. To avoid this, a current-limiting resistor may be required, although it may be the case that the intrinsic resistance of the diode is sufficient.

Using a further 50% margin of safety, any diode must then be rated to tolerate 18 V in the reverse direction. Whether or not a current-limiting resistor is used, the diode must also be rated both for the current and the power dissipation in the diode as a result of the current at 18 V.

In selecting a diode, TVS diodes are discounted for the reason that they are generally designed and specified for transients—very short duration spikes in voltage—rather than for continuous conduction, and, given the variability of the of the supply voltage (with magnitudes in the range 11.4–12.6 V being acceptable), it is possible that a diode rated for reverse break down at 12.3 V could conduct continuously, albeit at only a slight voltage of 0.3 V (Walters, 2010). The diode selected for use in the snubber must therefore be a Zener diode.

Using the parametric search functions provided by two online parts retailers, and specifying a Zener voltage of 12.0–12.55 V (to avoid reverse conduction at the nominal 12 V), the diodes given in Table 9.

Table 9

Shortlist of Zener diodes for use in a snubber circuit

Part No.	Zener Voltage (V)[13]	Zener Current (mA)[14]	Power (mW)	Th. Resistance to Air	Cost (£)
Diodes Inc. DDZ13ASF-7	12.43	10	500	250	0.130
Nexperia PDZ12B	11.99	0.5	400	340	0.176
Nexperia BZT52H-B12	12.00	1	375	330	0.222
Nexperia TDZ12J	12.00	1	500	250	0.284
Comchip CZRU52C13	12.00	1	150	—	0.299
Rohm VDZ12B	11.99	5	100	—	0.324

Nearly all Zener diodes returned by the parametric search—including many not listed in Table 9—were rated for too low a Zener voltage (often in the range 11.4–12.7 V, or 11.8–12.2 V), or were rated at too wide a tolerance (several diodes from Rohm were rated at 12–13.5 V, for example).

The Diodes Inc. DDZ13ASF-7 is very nearly the best choice—it offers a relatively high power rating, a relatively low case–air thermal resistance, and it is the least expensive. However, the tolerance for its Zener voltage, at 12.11–12.75 V, places it slightly outside (by 0.15 V) the rated maximum for computer fans. It therefore cannot be selected.

The remaining devices in the table offer Zener voltages in the range 11.74–12.24 V (for devices with a 11.99 V Zener voltage) or 11.8–12.2 V (for devices with a 12 V Zener voltage). This means that, potentially, 12 V might never be delivered to a computer fan, but does place the maximum voltage well within tolerance. Of the devices remaining, the Nexperia TDZ12J offers the best combination of minimum Zener current, power rating, thermal resistance, and Zener voltage tolerance, but does so at a relatively high price. However, in absolute terms, the price is only higher by around 9 pence per unit in single quantities, and would decrease when ordering in larger quantities. The device is also available (at a comparable price) at both retailers searched. The Nexperia TDZ12J is therefore selected as the Zener diode for use in the snubber circuit.

Next there must be selected the current-limiting resistor which will prevent the

[13] Where a range of voltages is given, this is the midpoint. Where possible, the minimum and maximum Zener voltages are within the range 12.1–12.6 V.

[14] Most data sheets did not specify a value for I_{ZK} (minimum Zener current to assure reasonable regulation) (ON Semiconductor, 2005, p. 7), and so where no value was specified the lowest Zener current rating given in the datasheet is taken to be I_{ZK}, and that I_{ZK} is the value represented in this table.

current through the diode from being such that the maximum power dissipation or maximum temperature of the diode is exceeded. While the ideal case would be to exceed the Zener current at all points during operation, it is not expected that this will be possible. However, by selecting the resistor to provide the maximum current for dissipating the maximum power at the maximum expected voltage, the best coverage across the range can be achieved. That maximum can be determined through application of the formula P=IV as follows.

$P = 0.5\ \text{W}$
$V = 18\ \text{V}$

$$\begin{aligned} P &= IV \\ I &= P/V \\ &= 0.5 / 18 \\ &= 0.0278 \\ I &= 27.8\ \text{mA} \end{aligned}$$

However, the ratings of the diode cannot be taken in isolation. The diode is specified as having a maximum junction temperature of 150 °C and a case–air thermal resistance of 250 °C/W (Nexperia, 2011b, p. 3). Therefore, at 0.5 W, the environment the diode is in must have an ambient temperature not exceeding 150−(250×0.5) = 150−125 = 25 °C. Unfortunately, the inside of a computer chassis is likely to have a widely-varying ambient temperature, and so it is unlikely that this 25 °C maximum could be attained. As well as a typical room temperature of 22 °C, a chassis could contain various high-power components exhausting heated air, and poor maintenance could lead to exhaust from the chassis being impeded by dust or other detritus. To gather information on the likely ambient temperatures, a computer was placed under load and temperature information provided by sensors mounted on the computer's motherboard were recorded. While portions of the system (in particular, around the computer's central processing unit) reached as high as 60 °C, neither of two "thermal zone" sensors on the motherboard exceeded 30 °C. Given the variability and unpredictability of this factor, the use of a 150% margin of safety is considered appropriate, and so the ambient temperature inside a computer chassis is assumed to be 75 °C.

Given the device maximum junction temperature of 150 °C, the power dissipated by the diode must then not exceed (150−75)/250 = 0.3 W. Performing the current calculation above with this new maximum power rating, it can be determined that the current through the diode must not exceed 16.7 mA. The minimum resistance can then be determined through application of Ohm's law V=IR as follows.

$V = 18\ \text{V}$
$I = 16.7\times10^{-3}\ \text{A}$

$$\begin{aligned} V &= IR \\ R &= V/I \\ &= 18 / 16.7\times10^{-3} \\ &= 1078 \\ R &\geq 1078\ \Omega \end{aligned}$$

Taking into consideration that the Zener voltage of the diode may be as high as 12.2 V, and that the supply voltage may be as high as 12.6 V, a worst-case Zener current can be calculated by further application of Ohm's law. If a 1100-ohm resistor were to be used (as it is highly unlikely any manufacturer sells a 1078-ohm resistor), the voltage of 0.4 V across the diode would produce a 0.36 mA Zener current. While this current is below the lowest testing value given in the datasheet (1 mA), the difference is relatively slight. It is expected that the Zener diode will conduct as required at the current level, albeit with some drift from the nominal Zener voltage. To be certain, this behaviour must be confirmed by practical experiment, if possible. Further, while there do exist other conditions which could result in a lower voltage across the diode (for example, a 12.2 V Zener voltage and 12.3 V supply voltage), under these conditions the supply voltage remains within the rated maximum for the fans and so it is immaterial whether the diode conducts or not.

Control transistor

In order to produce the PWM waveform that is filtered, there must be some way to switch that circuit. As the 12 V supply voltage exceeds the maximum ratings for the selected microcontrollers—although even if it did not, the microcontroller would not be able to source sufficient current—this control must be through a separate device. In this case, given the desired 100 kHz switching frequency, the selected device is a transistor.

As established, the transistor must be minimally rated to switch 1.5 A steady-state and 2.2 A pulsed at 12 V, and should be rated such that the power dissipated in it does not result in the junction temperature exceeding the maximum, taking into consideration the 75 °C ambient established earlier in this chapter. A parametric search of two online parts retailers was made, entering as many of these criteria as possible. In this search, a 50% margin of safety was added to each rating but the temperature. A shortlist of results is given in Table 10.

Table 10

Shortlist of PWM DAC control transistors

Part No.	**Drain-to-Source Voltage (V_{DSS})**	**Drain Current (A)**	**Power (W)**	**Thermal Resistance**	**Junction Temp. (°C)**	**Type**	**Cost (£)**
`Fairchild FDMS7682`	30.0	16.0	2.5	50	150	M	0.350
`TI CSD15571Q2`	20.0	10.0	2.5	50	150	M	0.360
`Alpha&Omega AOD514`	30.0	36.0	25.0	50	175	M	0.380
`WeEn PHD13005`	400.0	4.0	75.0	60	150	B	0.430
`ST BUL128D-B`	400.0	4.0	70.0	62.5	150	B	0.450
`Infineon IRF8707TRPBF`	30.0	9.1	1.6	50	150	M	0.450
`TI CSD17578Q3A`	30.0	20.0	2.5	50	150	M	0.510
`Infineon IPD30N06S2L-23`	55.0	30.0	100.0	50	175	M	0.750
`Infineon IGD06N60TATMA1`	600.0	6.0	88.0	62	175	I	0.790

Part No.	Drain-to-Source Voltage (V_{DSS})	Drain Current (A)	Power (W)	Thermal Resistance	Junction Temp. (°C)	Type	Cost (£)
ST STGP5H60DF	600.0	5.0	24	62.5	175	I	0.870
ST STL58N3LLH5	30.0	56.0	62.5	31.3	175	M	0.930

Note 1: The type abbreviations are B for BJT, I for IGBT, and M for MOSFET.

Note 2: The thermal resistances are junction–air or junction–mounting, depending on the information given in the relevant datasheet. The values given may only be attainable with the use of a copper-pour heatsink on the circuit board.

Considering that all devices listed in Table 10 have suitable voltage and current ratings, and that the power rating is largely dependent on temperature, a simple method of sorting the shortlist for selection is to sort for the most economical part based on that part's thermal characteristics. To do this, a score for each device was calculated as a quotient, where the thermal conductivity (i.e. the inverse thermal resistance, or W/°C) is the dividend and the cost is the divisor. A score so calculated increases with lowered resistance and decreases with greater cost.

The highest-scoring device is the Fairchild FDMS7682. Per the datasheet (2015a), the device is rated for a maximum junction temperature of 150 °C, and a thermal resistance of 50 °C/W when mounted on a 1 in^2 (approximately 6.5 cm^2) cooling pad on the circuit board. Adjusting for the ambient temperature of 75 °C, the limit for power dissipation is (150−75) / 50 = 75/50 = 1.5 W.

To determine whether the power dissipated by the FDMS7682 will remain within tolerance, datasheet figure 1 (the drain current against the drain–source voltage) can be applied to give a voltage drop of approximately 0.05 V at 3.3 A drain current (assuming gate–source voltage is 3.5 V), or an effective $R_{DS(ON)}$ of 15.2 milliohms—not an unreasonable value given the 8 milliohms listed for a gate–source voltage of 4.5 V and a drain current of 11 A. By applying datasheet figure 3 (normalised on resistance vs. junction temperature) for a worst-case scenario junction temperature of 150 °C, this effective resistance (and hence the voltage drop across the transistor) may increase by factor of around 1.55, and so $0.0152\times1.55=0.0235$ ohm is the effective worst-case scenario $R_{DS(ON)}$. This new effective resistance can then be substituted into the resistive power dissipation formula $P=I^2R_{DS(ON)}$ with the peak 3.3 A fan current to give the resistive power dissipation of $3.3^2\times0.0235=0.256$ W.

However, this is not the total dissipation—there are further losses involved with switching the transistor, calculated as $P=C_{RSS}V_{IN}{}^2f_{SW}I_{LOAD}/I_{GATE}$ per Keagy (2002). With a 5 mA gate current—selected to be well within the sourcing capacity of the selected microcontroller—this calculation gives a further dissipation of $(75\times10^{-12}\times12^2\times100\times10^3\times3.3)\ /\ (5\times10^{-3}) = 0.713$ W. The total power dissipation in the device is therefore $0.256+0.713=0.969$ W. This is well within the 1.5 W steady-state limit, and even further within any pulsed limit. The peak current may only be drawn for 1 second, and application of datasheet figure 13 (junction-to-ambient transient thermal response curve) shows that the effective thermal resistance for a one-second pulse does not exceed 0.6× the steady-state value. At a thermal resistance of 0.6×50=30 °C/W, the device maximum power dissipation can be taken as 2.5 W—not far from three times the calculated peak power.

If both the resistive and switching loss calculations were to be performed again with a current of 2.25 A (that is, the 1.5 A maximum continuous load current with a 50% margin of safety), the result would be $P_R+P_S=0.119+0.486=0.605$ W. This figure is even further below the steady-state safe maximum of 1.5 W, and so there is no issue with use of the FDMS7682.

The Fairchild FDMS7682 is therefore selected as the control transistor.

11. Driver stack

11.1 Driver fundamentals and stack selection

In the most popular computer operating systems, there exists a separation of privileges. Not at the application program level, where a program may execute as a normal user or administrator (or superuser), but at a more basic level—enforced by the central processing unit (CPU), software generally executes either in a high-privilege "kernel mode" or a low-privilege "user mode."

The primary use for this separation is security. In user mode, access to system memory is restricted, and certain instructions—if fetched by the CPU—fail to execute. In particular, most or all of the instructions related to input and output, to interacting with hardware devices connected to the CPU, are privileged. These restrictions prevent one program from interfering with another, and prevent any malicious software from interacting with—and potentially damaging—any device connected to the system.

In kernel mode (so called because it is the mode in which the core components of an operating system execute), access is entirely unlimited. Memory access is unrestricted, and kernel-mode instructions can modify the memory of processes which, in user mode, would be entirely isolated from one another. Most importantly for the purposes of this project, kernel-mode software can interface directly with any connected hardware devices.

Most modern operating systems permit device drivers to operate in kernel or user mode—either with direct hardware access in kernel mode, or by a user-mode driver communicating with a kernel-mode intermediary. Each method has its advantages and disadvantages and, while it is important that these are considered, a driver could be implemented as either type.

The obvious advantage of a kernel mode driver is that access is unlimited, and so any "special" device which needs particular handling can be supported. However, the primary disadvantages are complexity (in the kernel, access to the typical libraries which would be found in user mode is limited) and the unsafety—any fault in a kernel-mode driver has the potential to do widespread harm, whether by corrupting the memory of user-mode processes, introducing bugs which enable arbitrary code execution in kernel mode (as a worst-case example), or the simple fact that many things which would result in a user-mode process crashing will, in kernel mode, produce a processor exception and result in the operating system crashing (potentially resulting in data loss or corruption).

In contrast, a user-mode driver can make use of all the safety mechanisms afforded to regular user-mode programs, and is largely no more complicated than a regular user-mode program. The primary disadvantage is that a user-mode driver is limited to the functionality provided by its intermediary, and so any special or bespoke handling cannot be done by the user-mode driver. For the best of both worlds, many drivers are implemented as a hybrid system with the core functionality in a kernel-mode driver which is then utilised by a user-mode driver to provide further functionality.

Fortunately, the choice for this project is simple. As, for simplicity and ease of availability, the target computer uses a Windows operating system, the generic user-mode USB driver for Windows—WinUSB—can be used to interface with the fan controller.

11.2 Firmware considerations

Windows supports the automatic loading of WinUSB as the driver stack. This simplifies the installation process, and requires only that the firmware provide a specific set of values when queried. Such devices are referred to by the driver documentation as "WinUSB devices" (Microsoft Corporation, n.d.).

12. Control modes

One of the most fundamental features of the fan controller is the option to determine how the controller will behave in controlling the fans. While one mode of control may be acceptable, the ideal case is to provide several options which cover the probable needs of a majority of users.

This chapter will discuss a number of potential control modes.

12.1 Voltage control

Likely the simplest mode, enabling a user to control the fans through adjustment of the supplied voltage is also the crudest mode. A user would be provided with an interface that would enable the setting of fan voltage against temperature (or another set of data the controller collects or with which it is provided).

Control by voltage would depend heavily on the characteristics of the fan—for each fan there is a voltage below which the fan motor may fail to turn, and so the user would be restricted to a relatively thin band of voltages between 12 V and this minimum; and as discussed in chapter 8.2, the current drawn by the fan is expected to increase as the voltage supplied to the fan is decreased, and so the thin band of voltages would be further limited to ensure that component current ratings are not exceeded.

This mode is not unviable, but would require the fan controller to determine the characteristics of each fan. However, while this may seem like substantial work, it will be seen in the following sections that this requirement is independent of the control mode.

12.2 Speed control

Although somewhat analogous to control by voltage, speed control of fans is a layer abstracted. A user would continue to be provided with an interface enabling configuration against temperature (or other information), but instead of voltage the configurable parameter would be fan speed—likely as a percentage, but it would not be infeasible to give an absolute value in revolutions per minute.

The primary difference from control by voltage is that the controller would need to continually adjust either the voltage supplied to the fan or, for 4-pin varieties of fan, the duty cycle of the PWM signal provided to the fan. This would require that the controller continually monitor the fan tachometer output, and so this mode of control cannot be made available for 2-pin fan varieties (which lack a tachometer connection). Further, depending on the performance and viability of the voltage control discussed in chapter 10.2, it may be the case that the output of the tachometer for 3-pin fans is unusable, and so this mode of control may only be viable for fans of the 4-pin type.

If speed control of 3-pin fan types is viable as-discussed, then the issues that exist for voltage control also exist for the speed control of 3-pin fans. The controller must then determine the minimum starting voltage for each 3-pin fan, and must ensure that no change in voltage causes the current draw of the fan to exceed the 1.5 A permitted maximum.

12.3 Temperature point control

Control based around a temperature setpoint is the most abstract of the control modes considered. The mode is largely declarative, with the user specifying only a desired temperature, in contrast to the imperative voltage- and speed-control modes where the user must determine the correct fan configurations to attain a desired temperature—put more briefly, the mode specifies *what* instead of *how*.

While the concept is simple to understand—the further above the setpoint the temperature is, the faster the fans must spin—there are slight details important to the operation of the system. For example, the correct sampling rate must be chosen—too fast a rate, and there may be insufficient time for the temperature in the computer chassis to change, prompting the controller to rapidly increase the fan speed until it entered saturation; and too slow a rate could result in heating past what is desired as the controller fails to respond promptly.

Further, there may not be a “one size fits all” formula for the implementing of temperature setpoint control. One user may prefer if fans are powered off entirely until the setpoint is exceeded, while another may be indifferent. There is unlikely to be sufficient time to implement a highly-configurable type of setpoint control, but it is important that its potential future addition be taken into consideration in specifying the fan control protocol.

13. Protocol

The protocol defines the interface and rules for communication between the fan controller and the host computer. This chapter provides general discussion about the protocol and the specification of protocols for communication over USB. The protocol specification is given in Appendix D.

Per chapter 10.1, USB 2.0 is the revision in use. Nothing in this chapter or the protocol specification should be taken otherwise than in the context of USB 2.0 unless so identified.

13.1 Identification

While the most basic details of how a USB host identifies connected devices are not important here, it is important to be aware of the higher-level details of the communications between USB device and host, beginning with how a host selects the driver to be used to communicate with a USB device.

The most basic component of a USB device protocol is the endpoint. It may at first seem intuitive to think of an endpoint as being analogous to an Internet Protocol (IP) address—broadly, both identify the destination of a particular communication—but, in fact, an endpoint is more comparable to a Transmission Control Protocol (TCP) port in that an endpoint identifies a particular "service" of a USB device,[15] and a TCP port generally identifies a particular software application on a network host. This comparison is not perfect, and there is the facility for a transmission to be marked as being for an endpoint, an interface, or the USB device generally. However, despite any marking, the transmission is still made to an endpoint, albeit the specially-designated "default control" endpoint. To use the Open Systems Interconnection (OSI) model, an endpoint could be considered a constituent in the transport layer—the layer which "provides transparent transfer of data between session-entities and relieves them from any concern with the detailed way in which reliable and cost-effective transfer of data is achieved" (BSI, 1995, p. 32).

Each endpoint is a constituent, with or without other endpoints, of an interface. Where an endpoint represents a particular "service," an interface represents a particular facility provided by the device. For example, an interface might represent a printer, and that interface might have two endpoints—one endpoint for control, such as instructing the printer to begin a print, and another to transfer the data to be printed. A device may have multiple interfaces, each representing a different facility. For example, if the printer had built into it a port for a storage device (such as an SD card), it might have a separate interface for interaction with that port. Each interface may also have a list of alternates, which may each have a distinct set of endpoints. For example, a signal generator with a single output might define a single interface to control that output with alternate interfaces for each type of waveform the generator can produce—one alternate for a sine wave, one for a sawtooth wave, another for a square wave, and so on.

One or more interfaces is then grouped into a configuration, and a USB device may have multiple configurations. Unlike interfaces, only a single configuration is permitted to be active at any one time.

Associated with each of these logical groupings—configurations, interfaces, and endpoints—is a descriptor giving information about that grouping. Identification of a USB device primarily works with the interface descriptors or with a "device descriptor" containing device-wide information. A device or interface indicates its type (or class) through three fields—a class identifier, a subclass identifier, and a protocol revision identifier. These fields are generally the fields used by a host computer to identify which drivers to use.

[15] In the USB standard, the term *function* means "a USB device that provides a capability to the host, such as an ISDN connection, a digital microphone, or speakers." (p. 6). To prevent confusion, use of this term in any USB discussion has been avoided.

The standard class identifiers for generic classes of device are given in Table 11.

Table 11

Standard USB class identifiers

ID	Type	Description
`00h`	Device	Use class information in the Interface descriptors.
`01h`	Interface	Audio
`02h`	Both	Communications and CDC Control
`03h`	Interface	HID (Human Interface Device)
`05h`	Interface	Physical
`06h`	Interface	Image
`07h`	Interface	Printer
`08h`	Interface	Mass Storage
`09h`	Device	Hub
`0Ah`	Interface	CDC-Data
`0Bh`	Interface	Smart Card
`0Dh`	Interface	Content Security
`0Eh`	Interface	Video
`0Fh`	Interface	Personal Healthcare
`10h`	Interface	Audio/Video Devices
`11h`	Device	Billboard
`12h`	Interface	USB Type-C Bridge
`DCh`	Both	Diagnostic Device
`E0h`	Interface	Wireless Controller
`EFh`	Both	Miscellaneous
`FEh`	Interface	Application Specific
`FFh`	Both	Vendor Specific

For most of the classes, there exists a specification describing the subclasses and protocols to be used with that class. For example, the Mass Storage device class specification (USB-IF, 2010) defines a number of subclasses for the commands supported by storage devices—for example, subclass `06h` is for the SCSI command set, `08h` is for IEEE 1667 ("standard for discovery, authentication, and authorization in host attachments of storage devices"), and `FFh` is for vendor-specific command sets. The specification then uses the protocol identifier for the method of encapsulating this command set over USB. This generic specification would then allow an operating system to be shipped with a number of generic drivers, allowing basic devices (such as keyboards, mice, and storage devices) to function without the need to provide a driver of their own.

In the case of this project, however, there does not exist a relevant device class specification, and so the device would specify the "Vendor Specific" class. This is important for device identification in the drivers, although under the Windows operating systems a class of devices can be represented by a GUID[16] independent of the interface (i.e. whether USB, PS/2, or otherwise).

To give specific detail, software on the host computer uses a "device interface GUID" with the `SetupDiGetClassDevs` function to retrieve a handle[17] which can be used to retrieve a set of all devices connected to the host computer using that specific device interface GUID. In this case, the device interface GUID is specified in a Microsoft-defined descriptor the device returns when queried, and which is automatically registered with the operating system during USB device enumeration. The handle retrieved using that function is then provided to the `SetupDiEnumDeviceInterfaces` function, which enables basic information for each "device interface" to be retrieved. This basic information is then provided to the `SetupDiGetDeviceInterfaceDetail` function, which provides further detail on that specific device interface. The value returned from this function includes a "device path," which is provided to the `CreateFile` function as the file path argument. `CreateFile` returns a file handle, an abstraction for access to files and other input–output devices, which represents the USB device and which is provided to the `WinUsb_Initialize` function to retrieve a handle to the first USB interface reported by the device. This interface handle can then be provided as an argument to other functions in the WinUSB API to enable interaction with that interface—for example, the `WinUsb_GetDescriptor` function enables the retrieval of a device's descriptors given the interface handle, the descriptor type identifier, the descriptor index,[18] and language identifier (if the descriptor is a string descriptor), among other inputs (USB-IF, 2000, pp. 32–37, 244–245, 248, 261–263, 267–269; Axelson, 2009, pp. 103–112, 219–221; USB-IF, 2016; Microsoft Corporation, n.d.).

13.2 Communication

Modes of communication

While the previous section made a comparison to the OSI transport layer, it is not so easy to make a comparison to higher-level layers in the OSI model. Though what the USB specification refers to as *transfer types* most closely resemble the transport layer, the choice of transfer type impacts the choice of protocol in a higher layer. If an endpoint is set to be a *control transfer* endpoint, the use of a USB-defined request–response model is required, and a particular sequence of packets—a packet initiating the transfer, followed by a packet specifying the type of request, then by any data, and then by a status-reporting packet—must be sent, whereas with an *interrupt*, *bulk*, or *isochronous* endpoint, there exists no restriction on the communications model and any data sent in a transaction is

[16] Globally Unique Identifiers (GUIDs) are identifiers, also called Universally Unique Identifiers (UUIDs), generated in such a way that there is unlikely to be a conflict with another GUID/UUID generated by another person (ITU, 2012).

[17] A "handle" is an identifier specific to a particular application programming interface (API). A handle is analogous to a pointer in that a handle refers to a specific object and is used as an indicator where that object is stored. However, details of the interpretation of the value of the handle are generally known only by the API.

[18] The descriptor index is used to differentiate between multiple descriptors of the same type (USB-IF, 2000, p. 253).

transferred immediately after the initiator packet. Where in isolation the request–response model for control transfers could be considered part of the session layer—that is, the layer enabling entities "to organize and synchronize their dialogue and manage their data exchange" (BSI, 1995, p. 31)—or another higher layer, its relationship with what could be considered the transport-layer components of the system place it somewhere in-between.

This consideration out of the way, the transfer types are the main differentiators between endpoints. As well as enforcing (or not enforcing) a particular format for transmissions, each type defines a quality of service the host must afford to the device, and which the device can assume will be provided. The USB specification summarises the transfer types as follows (2000, p. 37):

- **Control transfers** — bursty, non-periodic, host software-initiated request–response communication, typically used for command–status operations.
- **Isochronous transfers** — Periodic, continuous communication between host and device, typically used for time-relevant information. This transfer type also preserves the concept of time encapsulated in the data. This does not imply, however, that the delivery needs of such data are always time-critical.
- **Interrupt transfers** — Low-frequency, bounded-latency communications.
- **Bulk transfers** — Non-periodic, large-packet bursty communications, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

Considering the requirements for the fan controller (chapter 2), it is unlikely that the fan controller will require the use of "any available bandwidth," except in the fulfilment of requirement 2.2.4 (the in-circuit updating of firmware). But, as established in chapter 9.4, there exists a separate device class specification for use where the ability to update the firmware of a USB device is desired. There is therefore little need for a bulk transfer endpoint.

Further, while communication between the fan controller and host computer is likely to be low frequency, there is no need for the maximum limit on latency that the interrupt transfer type provides. Regardless, the facility to arrange—in a standard way—for the periodic polling of a specific endpoint could be useful for any continuous status-monitoring process on the host computer. The use of an interrupt endpoint, perhaps in an alternate interface so that a monitoring facility can be enabled and disabled, is therefore something to consider, but provides no real advantage. Further, as per the USB specification (2000, p. 51), the interval specified for an interrupt endpoint is a maximum—the device could be polled once or many times in the specified period.

For much the same reasons as for interrupt transfers—and for the reason that a prerequisite is a Full Speed rather than Low Speed device—there is no benefit to be had from the use of isochronous transfers with the fan controller.

It therefore makes the most sense to use only a control transfer endpoint and, as the USB specification requires that endpoint zero be allocated by a device for use as the “default control pipe,”[19] use of only a control transfer endpoint has the advantage of requiring no additional endpoint configuration. Accordingly, any protocol defined for the fan controller must adhere to the “message pipe” format used by control transfer endpoints (USB-IF, 2000, pp. 34, 37, 44, § 8.5).

Control transfer specifics

In the terms used by the USB specification, control transfer pipes are “message pipes,” and must follow the standard format for requests and responses. A control transfer has three discrete stages—the setup, data transfer, and status. Entry into the setup stage is prompted by the host computer sending a SETUP packet, and it is this packet which contains the meta-information about the message. The format of this packet is shown in Table 12.

Table 12
Format of a USB SETUP packet
(USB-IF, 2000, table 9-2)

Offset	Field	Size	Value	Description
0	`bmRequestType`	1	Bitmap	Characteristics of request.
1	`bRequest`	1	Value	Specific request code.
2	`wValue`	2	Value	Word-sized field that varies according to request.
4	`wIndex`	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset.
6	`wLength`	2	Count	Number of bytes to transfer if there is a data stage.

The `bmRequestType` field of the packet specifies the data transfer direction (into or out from the host), the type of request (whether standard, specific to the device class, or specific to the vendor), and the recipient of the request (the device, an interface, or an endpoint). Qualified by this value, the `bRequest` field is used to identify the particular request being sent.

The USB specification defines a number of standard device requests which are defined on the device level and generally for use in the initial configuration of the device. The standard request of most interest in the specifying of the protocol is the `GET_DESCRIPTOR` request, as this request provides a standard method of retrieving class- or vendor-specific descriptors and so has the advantage that any function provided with a USB driver library for the retrieval of descriptors can be used, such as the `WinUsb_GetDescriptor` function.

The standard format for a descriptor is a single-byte field giving the length of the descriptor and a further single-byte field giving the type. In a manner similar to that for requests, the type-identifying byte is subdivided into further fields as a

[19] “Pipe” being the term the USB specification (2000, § 5.3.2) uses for an endpoint associated with software on the host computer.

simple method of dividing the available descriptor identifiers into standard, class-specific, and vendor-specific ranges. The format of the type-identifying byte is given in Table 13.

Table 13

Format of the USB `bDescriptorType` field

(USB-IF, 1997, § 3.11)

Bit	Field	Description
0	Identifier	The specific descriptor. For standard descriptors and descriptors for standard device classes, assigned by the USB-IF. For vendor descriptors, assigned by the vendor.
1		
2		
3		
4		
5	Type	The type of descriptor. 0 = Standard 1 = Class 2 = Vendor 3 = Reserved
6		
7	—	Reserved for future use. Reset to zero.

Where possible, then, the use of class- or vendor-specific descriptors should be considered over the use of class- or vendor-specific requests to avoid any unnecessary feature duplication (USB-IF, 2000, pp. 34–38, 225–228, 248–261, 269–273).

• PAGE INTENTIONALLY LEFT BLANK •

Design and Implementation

14. Proof-of-concept prototype

As noted at various points in the *Research and Theory* partition, it is important that aspects of the fan controller are practically tested and proven to work before any final prototype is designed or manufactured. If this were not done, the project could be irrecoverably compromised if it becomes clear that those aspects were not viable or required further work.

The two aspects of the controller requiring investigation are the following:

- Whether the PWM DAC functions as expected and, if it does, whether the output of the fan tachometer is accurate and usable at reduced voltage.
- What impact on the output of the fan tachometer there is when using PWM to directly modulate—that is, without the PWM DAC—the supply to the fan.

This chapter covers the design and development of a proof-of-concept prototype to investigate these aspects of the controller.

Appendix F contains resources relevant to the proof-of-concept prototype.

14.1 Hardware selection

The components selected in chapter 10 are to be used in the prototype. Covered in this section is the selection of ancillary components. Appendix F1 contains a final bill of materials for the prototype.

Flyback diode

The flyback diode enables the recirculation of current in the circuit when the control transistor is switched off. If the flyback diode were not included, the inductor's tendency to oppose changes in current would result in an increasingly high voltage being produced. Instead of producing this high voltage in an attempt to conduct through whichever medium (air, solder mask, et cetera), the inductor finds a relatively low-resistance path through the flyback diode.

The absolute minimum requirements for the flyback diode are largely the same as for other circuit components—a sustained current of 1.5 A, a pulse current of 2.2 A, a voltage of 24.4 V, and an ambient temperature of 75 °C. The temperature includes a margin of safety, but at least a 50% margin of safety must be added to the other requirements. A flyback diode must therefore be rated around 2.25 A continuously, 3.3 A peak, and 36.2 V. As before, a parametric search of the stock of two online parts suppliers was performed to identify candidate diodes.

Table 14

Shortlist of PWM DAC flyback diodes

Part No.	Forward Voltage (V)	Reverse Voltage (V)	Current (A)	Thermal Resistance	Junction Temp. (°C)	Cost (£)
`Rectron 1N5404-B`	1.00	400	3.0	30	150	0.185
`Comchip 1N5401-G`	1.00	400	3.0	18	125	0.306
`Diodes SDT5H100P5-7`	0.62	100	5.0	88	150	0.337
`Comchip CDBA540-HF`	0.55	40	5.0	24	150	0.352
`Nexperia PMEG4050EP`	0.49	40	5.0	120	150	0.360
`Diodes MBR5200VPTR-E1`	0.95	200	5.0	30	150	0.391
`Vishay SB550-E3/54`	0.65	50	5.0	25	150	0.406
`Diodes SB540-T`	0.55	40	5.0	25	125	0.498
`WeEn NXPS20H100C`	0.64	100	10.0	60	175	0.520
`ON NTSV30120CTG`	0.59	120	30.0	70	150	0.635
`Fairchild SB5100`	0.85	100	5.0	25	150	0.681

Note: The unit of thermal resistance is °C/W. The given thermal resistances may only be attainable in circumstances specified in the device datasheet.

The power dissipated by a diode is a function of the current through the diode and the voltage across the diode, and so a diode should have as low a forward voltage as is economically justifiable. Further, the rise in junction temperature that is experienced by a diode depends on the power it dissipates and its thermal resistance, and so thermal resistance should also play a factor. Therefore, in a manner comparable to that used to select a PWM DAC control transistor from Table 10, each device in Table 14 was assigned a score which decreased with forward voltage and price, and which increased with thermal conductivity.

The device with the highest score (0.168) is the Comchip CDBA540-HF. Using its forward voltage of 0.55 V and the maximum current of 3.3 A, it can be determined that this device will dissipate 0.55×3.3 = 1.82 W. With a junction–ambient thermal resistance of 24 °C/W, the diode temperature will rise by 1.82×24 = 43.6 °C. The ambient temperature is taken to be 75 °C and the device's maximum junction temperature is given as 150 °C, giving a maximum permissible rise in temperature of 150-75 = 75 °C. As the calculated 43.6 °C ≤ 75 °C, the Comchip CDBA540-HF can be used as the flyback diode for the filter.

Inductor

Unlike other components in the prototype, there are few factors which affect the selection of an appropriate inductor. Provided that the device is suitably rated for the most common parameters, the least-expensive option is often suitable for all but the most specialised of applications.

To select an appropriate inductor, a parametric search on the stock of two online parts retailers was performed. The parameters entered were a current rating of at least 3.3 A and an inductance in the 470–600 μH range. The first five lowest-cost results are given in Table 15.

Table 15

Shortlist of PWM DAC inductors

Part No.	Value (μH)	I_{MAX} (A)	Max. Temp. (°C)	Type	Cost (£)
Bourns PM2120-471K-RC	470±10%	3.3	105	SMT	1.840
Murata 60B474C	470±15%	3.5	85	SMT	2.270
J.W. Miller 2321-H-RC	560±15%	3.6	105	THT	2.440
Signal HCTI-560-3.6	560±15%	3.6	105	THT	2.530
J.W. Miller 2200LL-471-RC	470±20%	3.7	125	THT	2.910

In this case, the lowest-cost inductor is a highly-preferably option—the device has the tightest tolerance, a mid-range maximum temperature, and a satisfactory current rating. It is, however, a relatively large device, at approximately 3×2.5 cm, and so would consume considerable space on any circuit board. However, while the next device in the table (the Murata 60B474C) is provided in a 2.4×2.7 cm package, the low temperature rating precludes its use. The remaining devices are of a "through-hole technology" type, which can increase the cost of manufacture and assembly, and have looser tolerances for their values.

The temperature rating of the PM2120-471K-RC is relatively tight. The ambient temperature of 75 °C and the temperature rise above ambient of 30 °C typical at given in the device datasheet would result in the device operating at its absolute maximum rating if it were operated at 3.3 A. However, this 3.3 A is a peak current and includes a sizable margin of safety—at the 2.25 A continuous current (which also incorporates a margin of safety), the temperature rise will not be as great.

The Bourns PM2120-471K-RC is therefore a suitable device for use in the filter.

Capacitor

As with the choice of inductor, the least expensive of any appropriately-rated capacitor will generally be suitable for use. As before, a parametric search of the stock of two online parts retailers was performed with the parameters being a rated capacitance of 4.7 µF, a tolerance of 5–10%, a D.C. voltage rating of 50 V or greater, a surface-mount package type, and a maximum operating temperature of at least 125 °C. Table 16 lists the first four results with unique manufacturers and the first device rated for 150 °C.

Table 16

Shortlist of PWM DAC capacitors

Part No.	V_{DC}	Max. Temp. (°C)	Package	Cost (£)
Taiyo Yuden UMK316AB475KL-T	50	125	1206 (3216)	0.190
Samsung CL31B475KBHNNNE	50	125	1206 (3216)	0.250
KEMET C1206C475K5RACTU	50	125	1206 (3216)	0.270
Murata GRM21BC71J475KE11L	50	125	0805 (2012)	0.400
UCC KVF500L475M55NHT00	50	150	2220 (5750)	1.580

Note: Packages are given as inch code (metric code).

None of the capacitors given in Table 16 had datasheets which listed thermal data, and only the UCC capacitor datasheet included ripple current rating information. However, it is anticipated that the relatively wide margin of 50 °C above ambient for the absolute maximum operating temperature will be sufficient. If desired, a wider margin of 75 °C is possible with a capacitor rated for 150 °C, but this added tolerance comes at a significant price increase—nearly 4× the cost of the most expensive capacitor listed in the table with a 125 °C rating, and over 8.3× the cost of the least expensive capacitor in the table.

Any device in Table 16 is suitable, and so the least expensive can therefore be selected for use in the fan controller.

Tachometer interface

Fans of the 3- and 4-pin variety provide a tachometer for use in monitoring the speed of the fan. The tachometer is specified as an open-collector or open-drain output, meaning that the output is connected to ground through a transistor and that, when the transistor is not switched on, the output is left to float. Hence, the specification for 4-pin fans requires that a resistor to pull the voltage up to 12 V be included (Intel Corporation, 2005a, p. 9).

If a pull-up to 12 V were to be used, the tachometer and the microcontroller could not be directly connected—the development kit microcontroller's pins are rated for a maximum voltage not greater than 0.3 V above the microcontroller supply voltage, which equates to an absolute maximum voltage of 4.1 V (Silicon Labs, 2014b, p. 10). In order to connect the two without damaging the microcontroller, a level shifter would need to be placed between the microcontroller pin and the pulled-up tachometer signal. Any selected level shifter would be required to operate at the minimal current permitted through the pull-up resistor.

Alternatively, the tachometer pull-up could be connected to the 3.3 V supply pin available from the development kit. This would permit direct connection from the tachometer to the microcontroller, but would mean noncompliance with the fan specification. Although no potential issues are foreseen as a result of using a pull-up to a lower voltage, specification compliance is a higher priority than the marginal reduction in cost from not using a level shifter.

In selecting an appropriate level shifter, a simple parametric search was found to be unsuitable—neither of the online parts retailers consulted permitted searching by the direction of shifting (i.e. high-to-low or low-to-high), nor by whether any precise ordering in the connection of supplies was required (a common property of level shifters). Instead, a parametric search was used to identify level shifters with sufficient voltage ratings (i.e. 12.6 V, with no further margin of safety as the maximum voltage is specified by the PCI-SIG (2008)), and a manual evaluation of the results is made later in this section. Other than isolators, the only devices revealed by the search as being able to translate voltages of up to 12.6 V were various logic devices—specifically, the 744050, 4104, 4504, and 40109.

Isolators are generally intended for safety applications, and have a number of different operating models. Opto-isolators use light shone onto a photosensitive transistor to transmit a signal from one circuit to another, while magnetic isolators use a miniature transformer in combination with edge-detecting logic. Other types of isolator transmit signals using radio emissions (Krakauer, 2011). Complex construction and certification for use in safety-critical applications mean that an isolator will generally be more expensive than a level shifter—indeed, the least expensive isolator of those stocked by two online parts retailers was listed at a price of £0.59 in single quantities. Comparatively, the least expensive level shifter provided was listed at a price of £0.26 in single quantities. Each provided only a single input–output pair. Isolators, then, can be discounted unless there exists no level shifter capable of the translating the same voltage levels.

The 744050 is a “hex non-inverting buffer” capable of both low-to-high and high-to-low level shifting. High-speed CMOS versions (the 74HC4050) operate from a supply voltage of 2–6 V, but are rated for input voltages up to 15 V. A downside to this device is that, while it may be appropriately rated, the device package may cause issues in layout—each input pin is adjacent to its corresponding output, and so with any considerable number of connections it is likely that one wire would need to cross another. This is not a huge issue, but could consume more valuable circuit-board space than may be desirable (Texas Instruments, 2005a). This device is not to be confused with the 4050—the 4050 requires a 3–15 V supply, and is therefore less desirable where a 3.3 V supply may be the primary supply in use.

The 4104 is a “quad low-to-high voltage translator with 3-state outputs.” It is able to translate voltages up to 15 V, but only in the low-to-high direction—a 4104-type device datasheet notes that the input supply $V_{DD(A)}$ “must always be less than or equal to [the output supply] $V_{DD(B)}$, even during power turn-on and turn-off” (Nexperia, 2016c). This device is therefore unsuitable for use in this application.

The 4504 is a “hex non-inverting level shifter for TTL-to-CMOS or CMOS-to-CMOS." It can be used in either the high-to-low or low-to-high directions, and is rated for a 3–18 V supply voltage. Unlike the 744050 and 4104, the 4504 includes a mode selector (TTL–CMOS or CMOS–CMOS), which adjusts the tolerances of logic-high and logic-low voltages. For example, at 25 °C with an input supply V_{CC} of 5 V and output supply V_{DD} of 15 V, the maximum logic-low voltage is specified as 0.8 V when the device is in TTL–CMOS mode and 2.25 V when in the CMOS–CMOS mode (ON Semiconductor, 2014). As before, the device's requirements for the supply voltage may make the device less desirable as a result of the relatively tight margin between its 3 V minimum and the available 3.3 V-nominal supply.

The 40109 is a "quad low-to-high voltage level shifter" which can also operate as a high-to-low shifter. As the datasheet specifies, "[the input supply voltage] V_{CC} may exceed [the output supply voltage] V_{DD}, and input signals may exceed V_{CC} and V_{DD}. When operated in the mode $V_{CC} > V_{DD}$, the CD40109 will operate as a high-to-low level shifter." The 40109 does require that any input signal have a voltage of at least 0.7× V_{CC}, but otherwise there are "no restrictions on the relative magnitudes or input signals within the device maximum ratings." Further, unlike the 4104, there is no requirement for the supplies V_{CC} and V_{DD} to be connected in a specific order. Given that the maximum supply voltage rating is 18 V, the 40109 could be used as a high-to-low 12 V–3.3 V level shifter. However, as with the 4050 and 4504, the minimum recommended supply voltage of 3 V leaves little margin with the available 3.3 V supply (Texas Instruments, 2003). This makes the device less desirable as a choice but, as before, does not necessarily preclude its use for this application.

The choice of level shifter is then a choice between the 744050, the 4504, and the 40109. The 744050 is the most favourable option, offering the best tolerance for supply voltages for this application. The input voltage tolerances for the three devices are largely similar, at 15 V for the 744050 and 18 V for the 4504 and 40109. Unless it is significantly more expensive or significantly harder to procure than either of the 4504 or 40109, it is expected that the 744050 will be selected for use in the fan controller. To assess cost and availability, a search of the stock of two online parts retailers was made for 744050-, 4504-, and 40109-type devices. The results of the search are given in Table 17.

Table 17

Cost and availability of various level shifters

Part No.	Retailer 1		Retailer 2	
	Units	Cost (£)	Units	Cost (£)
TI CD40109BPWR	7363	0.310	2825	0.306
Toshiba 74HC4050D	4302	0.310	3654	0.474
TI CD74HC4050M96	17676	0.340	9226	0.329
Nexperia 74HC4050PW	4084	0.340	4539	0.337
TI CD4504BM96	25442	0.340	11414	0.337
ON MC14504BDG	1495	0.800	6810	0.819

Note: The "units" columns are the number of units the retailer listed as being ready to ship or ready for immediate dispatch on the day the search was made.

As can be seen, there is little real difference in price between the 744050, 4054, and 40109. As a result of its being more favourable in comparison to the other devices, the 744050 is selected as the device type, and the Texas Instruments CD74HC4050M96 (or equivalent) is selected as the particular make and model for use in the fan controller as a result of its pricing and availability.

Fan PWM interface

The speed of a fan of the 4-pin variety is controlled by varying the duty cycle of a PWM signal transmitted down the fourth fan pin. A fan is required to provide an internal pull-up to a voltage not exceeding 5.25 V, and a fan controller is to control the fan by providing an open-collector or open-drain output. This output must tolerate a minimum of 5 mA, with a recommended 8 mA tolerance. While an open-drain or open-collector output would pull the signal to ground, the specification states that the "signal is not inverted" and that "100% PWM results in max fan speed." This likely indicates that the fan includes the necessary logic to convert the signal as required, but whether this holds true should be confirmed by practical experiment (Intel Corporation, 2005a, pp. 9–10).

As established in the *Tachometer interface* section above, the microcontroller to be used in the fan controller is able to tolerate, under specific circumstances, an absolute maximum input voltage of 4.1 V. As a fan is permitted to pull the voltage up to 5.25 V, a microcontroller pin cannot be used directly. Instead, a further transistor must be used to provide a 5.25 V-tolerant open-collector or open-drain output. Although not the most economical option, no practical reason precludes the use of the Fairchild FDMS7682 selected earlier in chapter 10.2.

As established in chapter 10.2, the FDMS7682 has an $R_{DS(ON)}$ of 15.2 milliohms at an approximate gate–source voltage of 3.5 V. Using the power calculations in that chapter (where the frequency f_{SW} is taken as 28 kHz, the maximum acceptable fan PWM frequency), the resistive power dissipation in the transistor would be $I_{LOAD}^2R_{DS(ON)} = 0.008^2(15.2\times10^{-3}) = 0.973\ \mu W$. The switching loss in the transistor is $C_{RSS}V_{IN}^2f_{SW}I_{LOAD}/I_{GATE} = (75\times10^{-12}\times5.25^2\times28\times10^3\times8\times10^{-3}) / (5\times10^{-3}) = 92.6\ \mu W$, for a total dissipation equalling $0.973\ \mu W + 92.6\ \mu W = 92.7\ \mu W$. Taken together with the 125 °C/W thermal resistance when the device is not mounted on a copper pour heatsink, the expected temperature rise above ambient is less than a tenth of a degree Celsius.

Power connector

There must be a connector for supplying power to a connected fan, and while the selected PCI-E auxiliary power connector would provide sufficient power, it would not be the most suitable connector for a prototype. The connector for use in a prototype should enable the connection of almost any variety of supply, and hence a suitably-rated screw terminal should be used. A screw terminal requires only that a wire be connected and held in place by a screw, and so any power supply can be used—either directly if the supply also uses a terminal accepting a bare wire, or indirectly via a "crocodile clip" or similar connector with one end of the wire in the screw terminal and the other in the clip's jaws.

The same ratings as for other components apply largely unchanged to the power connector, except that the rated current must be doubled and the voltage rating need only be the maximum supply voltage plus a 50% margin of safety ($12.6\times1.5 = 18.9$ V). The two aspects to be investigated require separate circuit configurations, and so the connection of two fans at once to the prototype board is likely to be possible. The connector must therefore be rated to carry the current necessary to concurrently power to fans—6.6 A. In addition, the search was limited to two-position terminals. Using these parameters, a parametric search of two online parts retailers was made. The results are given in Table 18.

Table 18

Shortlist of proof-of-concept prototype screw terminals

Part No.	V_{MAX} (V)	I_{MAX} (A)	Cost (£)
Altech MBES-152	300	10.0	0.214
TE Connectivity 1776493-2	300	10.0	0.253
Phoenix Contact 1792863	400	12.0	0.310
Wurth Electronics 691137710002	300	16.0	0.380
On Shore Technology OSTTE020161	250	16.0	0.670
Amphenol VI0201550000G	300	16.0	0.750
Molex 0398890042	300	17.5	0.900

As can be seen, there is insignificant variation across a range of manufacturers and prices. As each is rated well above the supply voltage and considerably above the maximum supply current, the use of any connector given in the table is therefore suitable. The least expensive option may be passed over for a more expensive option from a larger manufacturer, as this would likely aid the ability to order in volume and decrease the likelihood of abrupt discontinuance of the connector.

Supply-monitoring transducers

In order to adjust the voltage supplied to a fan more accurately, to provide status updates to the host computer, and for reasons of safety, the fan controller must have means to monitor the voltage and current supplied to each fan. Voltage monitoring is relatively simple—the appropriate point in the circuit is connected, whether or not through a potential divider, to an analogue-to-digital converter. This converter then produces a value representing the voltage, which can be used directly in firmware.

Current sensing is not as straightforward—while monolithic current transducers are available, they are available at considerable cost. The least expensive current transducers stocked by two online parts retailers were priced at £1.39 (Allegro ACS711KEXLT-31AB-T) and £2.70 (Melexis MLX91209CA), respectively. A less expensive method is the use of a so-called “sense resistor,” where the voltage across a low-value resistance in series with the load is measured. With the known resistance and the measured voltage, the current can be found per Ohm’s law. In contrast to the monolithic current transducers mentioned above, the least expensive current-sense op-amps stocked by the same two online parts retailers were priced at £0.450 (TI INA180A2IDBVT) and £0.437 (Silicon Labs Si8540).

In terms of microcontroller resources, these monitoring capabilities are relatively expensive—to measure the analogue voltage produced by the transducer or op-amp, an analogue-to-digital converter (ADC) is required. This would mean that, on a final prototype, at least eight ADC channels are required (two per fan, one of which is for current and the other for voltage). The EFM32WG990 controller on the development kit provides an eight-channel ADC, and so there would be no ADC channels available for other inputs without further external multiplexing.

However, this use of microcontroller resources would remain the same whether a monolithic transducer or current-sense op-amp were used, and so any choice must be based on the monetary cost of the monitoring technique. As established, the use of a current-sense op-amp is likely to be less expensive, and so it is a current-sense op-amp which is to be used. In selecting an appropriate part, there are relatively few criteria—the op-amp must accept a supply voltage available on the prototype, it must output a voltage within the acceptable range for the input pins on the microcontroller, and it must be capable of measuring currents of either the range 0–3.3 A (if attached only to a single fan) or 0–6.6 A (if used to measure the total current supplied). As there are no other distinguishing factors, the least-expensive device with suitable ratings can be selected.

As the difference in pricing was minimal, and both retailers stocked this device, the Texas Instruments INA180A2IDBVT is selected for use.

For voltage monitoring, any suitable combination of resistors giving the correct ratio for potential division may be selected. Given that the fan supply voltage is not expected to exceed 12.6 V and given that the nominal supply voltage for the microcontroller is 3.3 V, a ratio of the fan supply to microcontroller supply voltages (that is, 12.6:3.3 ≈ 3.82:1) would—considering that the minimum voltage for the microcontroller to register a logic high is 0.7 V—enable measuring a fan voltage in the approximate range 2.67–12.6 V. While it is unlikely that a resistor that is a multiple of 3.82 will be found, 3.83 is in both the E48 and E96 standard series of resistors, and so a 3.83:1 ratio is possible (BSI, 2015). Provided that a ratio places the resultant voltage within the tolerable range and limits current sufficiently, the precise choice of values is unimportant—the microcontroller included with the development kit has a floating-point unit (FPU), and so there is little downside to using values which do not divide as cleanly. Using a 38.3 kiloohm and 10 kiloohm pair of resistors, the divider would be capable of safely measuring fan voltages in the range 2.68–12.64 V. The datasheet for the EFM32WG990 microcontroller used in the development kit (Silicon Labs, 2014b) does not specify a maximum input current, and so it is important that an especially conservative level be permitted. If resistors of 40 and 10 kiloohms were chosen, the maximum current into the pin would be 261 μA at 12.6 V. It is expected that this will be sufficiently low so as not to result in damage.

Temperature sensor

While the primary purpose of the proof-of-concept prototype is to enable testing of aspects of the final design, the inclusion a temperature sensor would result in the prototype largely mirroring the expected functionality of the final design. At minimal additional cost, the prototype could then be used as a platform for the development of the firmware and software for the controller, potentially saving time. Inclusion therefore makes practical and economic sense.

However, a simple temperature sensor which produces an analogue voltage output (such as a thermocouple) would require a further ADC input channel. As was established in *Supply-monitoring transducers* above, the final design will require eight ADC channels (the maximum provided by the microcontroller) and so a device which would require a further channel cannot be included. The microcontroller does include an internal sensor for the ADC, but this would read the temperature of the microcontroller rather than the temperature of the environment (Silicon Labs, 2014a, p. 692). The solution is, then, the use of a digital-output temperature sensor—specifically, the use of an I^2C temperature sensor, as a connection to an I^2C function is available from the development kit expansion header. To select an appropriate device, a parametric search of the stock of two online parts retailers was performed for a device which could communicate using an I^2C connection, which could be powered from a 3.3 V supply, and which came in a surface-mount style package. The results are given in Table 19.

Table 19

Shortlist of proof-of-concept prototype I^2C temperature sensors

Part No.	**Range (°C)**	**Accuracy (Typ.)**	**Cost (£)**
`Maxim DS75U-C12`	-55–125	±3.0 °C	0.490
`Microchip/Atmel AT30TS74-SS8M-T`	-55–125	±2.0 °C	0.513
`NXP PCT2075TP,147`	-25–100	±1.0 °C	0.530
`Microchip/Atmel AT30TS74-XM8M-B`	-55–125	±2.0 °C	0.551
`NXP LM75ADP,118`	-55–125	±3.0 °C	0.600
`ST STLM75M2F`	-55–125	±0.5 °C	0.690
`Microchip MCP9844T-BE/MNY`	-40–125	±1.0 °C	0.750
`TI TMP103AYFFR`	-40–125	±1.0 °C	0.880
`Silicon Labs Si7050-A20-IM`	-40–125	±1.0 °C	0.896
`Silicon Labs Si7051-A20-IM`	-40–125	±0.1 °C	1.820

In order to select the most economical temperature sensor, a score for each was computed by having the score increase with widening range and improving accuracy and decrease with cost. The Silicon Labs Si7051 device had the largest score (17.170), followed by the NXP PCT2075TP (at 7.547). While the Si7051 offers considerably greater accuracy, it is unlikely that accuracy better than 1.0 °C will be required, and so the NXP PCT2075TP is selected.

Fan connector

There is little selection involved for the fan connectors—as well as specifying the dimensions of the header, the specification for 4-pin fans also lists a number of known-compatible parts (Intel Corporation, 2005a, p. 19). The connectors for 2- and 3-pin fans are known to be compatible with the connector for 4-pin fans (per chapter 5.1). All that is required is then to search the stock of online parts retailers for the part numbers listed in the specification. The results of this search are given in Table 20.

Table 20

Fan header pricing

Part No.	Cost 1 (£)	Cost 2 (£)
`Foxconn HF27040-M1`	—	—
`Tyco 1470947-1`	—	—
`Wieson 2366C888-007`	—	—
`Molex 47053-1000`	0.320	0.337

Only the Molex 47053-1000 connector was stocked by the two online parts retailers the stocks of which were searched. While it would be preferable to have multiple known-compatible parts available, Molex is a company of considerable size and the Molex part is available from both retailers. There are unlikely to be concerns over the availability of parts, and so the 47053-1000 can be selected as the fan header for use in the fan controller.

Expansion header

There is also little selection involved for an appropriate female counterpart for the development kit's expansion header. Any header with 20 connections split into two rows of ten where the connections are 100 mil (2.54 mm) apart may be selected.

The Sullins Connector SFH11-PBPC-D10-RA-BK was selected as it was available from the prototype assembly house and was inexpensive (£1.07 in single units).

14.2 Microcontroller connection

The development kit to hand exposes a number of the pins of its microcontroller, both through bare conductive "breakout" pads and through a pin header. For ease of connection, the pin header is the most suitable choice for the connection of the development kit to the prototype board—no soldering is involved, the header provides a robust and secure means of connection, and the *de facto* standard pin pitch[20] of 100 mil (2.54 mm) means that counterpart female headers are likely to be widely available at low cost.

Each of the pins exposed (with the exception of the power pins) is multifunction, and firmware on the microcontroller can adjust the function assigned to the pin, but only a limited number of functions are available on any given pin. To ensure that the prototype can be correctly controlled, it is therefore necessary to have an exhaustive list of all required functions which can be cross-referenced with a list of all available functions. This exhaustive list then enables a correct scheme for connecting the microcontroller to the prototype to be produced before any physical circuit is designed.

The following functions are required for the prototype:

- **Four PWM functions**—two to be connected to the PWM control inputs for each fan connected to the prototype, one connected to the gate of the control transistor for the PWM DAC, and one connected to the gate of the

[20] "Pitch" refers to the distance between the centre points of each pin.

transistor used to directly modulate the supply voltage for the fan without the PWM DAC connected to it.

- **Four ADC channels**—two channels per fan, where for each fan one channel is to be used to measure the voltage provided to the fan and the other is to be used to measure the current (via a current transducer).
- **Two pulse-counter functions**—one for each of the fan tachometer output, although it would be possible to use a general-purpose pin instead. A pulse-counting function is preferable as it enables counting to be done by an autonomous peripheral and without CPU intervention, which is likely to reduce the complexity of device firmware.
- **One I^{2}C function**—to be used for the connection of a temperature sensor.

Using tables 4.1 and 4.2 of the datasheet for the EFM32WG990 microcontroller on the development kit (Silicon Labs, 2014b, pp. 58–70) with table 9.1 and figure 9.2 of the user manual for the development kit (Silicon Labs, 2013d, pp. 16–18), the pins available to the prototype can be mapped against the list of required functions above. This map is provided in Table 21.

Table 21

Map of required pin functions to available pins

MCU Pin	Header Pin	PWM	ADC	PCNT
PB11	11	2, 3	—	—
PB12	13	3	—	—
PC0	3	1	—	1
PC3	5	—	—	—
PC4	7	3	—	2
PC5	9	3	—	—
PC6	15	—	—	—
PD0	4	—	1	3
PD1	6	1	1	—
PD2	8	1	1	—
PD3	10	1	1	—
PD4	12	—	1	—
PD5	14	—	1	—
PD6	16	2, 3	1	1
PD7	17	2	1	—

Note: The PWM function numbered 3 is provided by the low-energy timer (LETIMER).

It can be seen that there is minimal overlap between functions. Accordingly, for the PWM functions pins PB11, PB12, PC0, and PD7 are preliminarily selected;[21] for the ADC functions, pins PD1 through PD4 are preliminarily selected; for the pulse-counting functions, pins PD0 and PD6 are preliminarily selected; and for the I^2C functions, pins PC4 and PC5 are preliminarily selected. It may be the case that these pin selections inconvenience routing signals on any circuit board design, in which case these preliminary selections may be adjusted.

14.3 Hardware design rationale

Appendix F2 contains the schematic diagrams representing the design of the proof-of-concept prototype, and Appendix F3 contains the circuit designs produced from the schematic diagrams in Appendix F2. Appendix F4 contains photographs of the manufactured circuit board.

Where it has been considered necessary or helpful, the rationale for certain decisions made in producing those diagrams and designs has been included in this section. An effort has been made to keep each rationale brief, and to keep all of the more detailed hardware selection within chapter 14.1.

Where possible, each of the rationales given in this section is given in the same order as the relevant portions of the schematic diagrams appear.

Locations in schematics are given as the sheet, column, and row. For example, on sheet five (5) the third (3) column and first (A) row would be 5.3A.

MOSFET gate resistor

The 470-ohm value for the resistors connected between the microcontroller and the MOSFET gate was selected to limit the current to the 5 mA drive current used in calculations in chapters 10.2 (p. 38) and 14.1 (p. 56) when the microcontroller output voltage is at the minimum 0.8× supply voltage permitted whilst configured to source a current of 20 mA with a 3.0 V supply (Silicon Labs, 2014b, p. 20).

While the 470-ohm value would result in 5.1 mA being sourced at 0.8×3.0 V and up to 7.02 mA being sourced at 3.3 V, neither value is outside the rated source current of the microcontroller pins, and neither is expected to adversely impact the heat dissipation of the MOSFETs. Any resistor selected for use must be rated to dissipate at least 24 mW.

Schematic locations 2.4B, 2.5C, 3.4C, and 3.5B.

PWM DAC voltage transducer

The voltage transducer for the PWM DAC is not connected between the 12 V input of the fan and ground, but between the 12 V input of the fan and the point between the PWM DAC’s filter capacitor and filter inductor. This mirrors the connection of the fan itself, as the fan is connected to ground through the filter and not directly. As measuring the voltage across the fan requires use of the same

[21] It should be noted that PB12 and PD7 are different output channels for the same timer, and so cannot be configured to provide differing-frequency PWM output (although the duty cycle remains independently configurable). However, as the channels are used in the control of separate fan connections and as simultaneous control of multiple fans is not required in the prototype, this is immaterial.

reference, the voltage transducer must also be connected in this way.

The same is not true of the bare fan connection, as its connection to ground (i.e. through a transistor) is, for this purpose, effectively the same as if it were directly connected. Connection through a transistor does mean that the transducer is not guaranteed to produce an accurate 0 V reading when the transistor is switched off, but the transistor being switched off removes any need for a voltage reading and so this effect is inconsequential.

Schematic location 2.2B.

Current-sense resistor

The current-sense resistor was chosen in accordance with section 9.1.2 of the datasheet for the INAx180 series of devices (Texas Instruments, 2017), which provides an equation to calculate the maximum value of the resistor R_{SENSE} given a maximum permitted power dissipation, and two equations for verifying that any chosen value is suitable for use.

The first equation is a simple rearrangement of Joule's law $P=I^2R$ which, given a maximum resistor power dissipation of 125 mW (a common value) and a load current of 3.3 A, indicates that the resistor cannot exceed 11.5 milliohms.

The other two equations are used to verify that the op-amp remains within its maximum output voltage swing range.[22] All devices in the INAx180 series are specified as having a maximum positive swing of 30 mV below supply, and a minimum negative swing of 5 mV above ground. The equations depend on the gain of the amplifier, which is listed as 50 for INAx180A2 devices (Texas Instruments, 2017, p. 6). Taking a value of 10 milliohms with a maximum load current of 3.3 A and a minimum load current of 25 mA (half of the minimum listed in Table 4), the equations indicate that a maximum positive swing of 1.65 V and a minimum negative swing of 12.5 mV will be produced. A 10-milliohm resistor is therefore a suitable choice for a current-sense resistor, provided that it is rated to dissipate at least 109 mW.

Schematic locations 2.3C and 3.2B.

MOSFET copper pours and stitched vias

In the circuit design, transistors M1 and M3 (those being the transistors which are connected in series with the fans and which switch to ground) are mounted on copper pours of approximate area 1 in^2 with vias[23] to ground around the edge.

The copper pour is recommended in the datasheet for the FDMS7682 to lower the junction–air thermal resistivity to 50 °C/W, and is necessary for operation at

[22] That range being the maximum range of voltages which "can be obtained without waveform clipping..." (Karki, 1998) or, to rephrase, the maximum and minimum voltages which the op-amp can produce on its output.

[23] A via being a hole in the circuit board that is plated on its inner surface. Vias enable a signal to be routed across multiple layers of a circuit board.

the power levels determined in chapter 10.2 (Fairchild, 2015a).

The vias serve multiple purposes—first, they provide an electrical connection to ground for the transistor; second, they thermally connect the copper pour heatsink to the ground pour, which should aid in removing heat from the device; and third, they act as a crude Faraday cage to mitigate noise transmission to or from the transistor.

14.4 Firmware

Program specification

Microcontroller firmware is required to operate the proof-of-concept prototype circuit. The firmware must have at least three modes of operation, as follows:

14.4.1 A mode where the fan connected via the PWM DAC is powered and the duty cycle of the signal into the PWM DAC can be varied. The variations may be fixed or determined by user input, and must minimally consist of a variation with a duty cycle of 100% and a variation with a duty cycle of less than 100%. The firmware must alternate between variations in response to user input.

14.4.2 A mode where the fan connected to the bare fan connection is powered and the supply line to the fan is modulated by PWM, where the duty cycle can be varied as in mode 14.4.1.

14.4.3 A mode where the fan connected to the bare fan connection is powered and the PWM control line to the fan is modulated by PWM, where the duty cycle can be varied as in mode 14.4.1.

The firmware must alternate between these modes in response to user input.

The firmware must report, on the development kit LCD or by other means, the current mode and duty cycle.

Design and source code

Appendix F5 contains resources and discussion produced in designing firmware for the proof-of-concept prototype. Appendix C2 contains source code for the firmware produced from the resources and discussion in Appendix F5.

Interrupt dependencies

Interrupts[24] on the microcontroller on the development kit generally require that the functions of generating an interrupt service request and servicing a request are configured separately—for example, a timer must be configured to generate a request at the expiry of a specified period, and the microcontroller's nested vectored interrupt controller (NVIC) must be configured both to acknowledge such requests and with information about a relevant service routine (Silicon Labs, 2014a, p. 13).

[24] An interrupt is an indication to the processor that an event external to the processor has occurred, or that a system external to the processor requires attention. A processor will generally respond to an interrupt request by pausing normal code execution and switching to an interrupt service routine. Once the interrupt request has been serviced, the processor returns to normal code execution.

The NVIC is a standard part of the ARM Cortex-M4 processor architecture used by the microcontroller, and is more fully described in the Cortex-M4 Devices Generic User Guide (ARM, 2010, ch. 4). Briefly, the NVIC supports selectively enabling and disabling interrupts, assigning interrupts a priority level (to enable the most important of a set of simultaneous interrupts to be serviced first), and both level-sensitive and pulse interrupt request signals. Interrupts can also be triggered by the firmware executing on the device,

This section serves as a reference for the interrupts required in the firmware (see chapter 14.2 for some sources), and lists the relevant interrupt request source, the NVIC interrupt number, and the source-specific registers for the configuration of interrupts for the given interrupt request source. This list is given in Table 22.

Table 22

Microcontroller interrupt dependencies

Peripheral	NVIC No.	MCU Registers
USB	5	`USB_CTRL, USB_STATUS`
I^{2}C Bus 1	10	`I2C1_CTRL, I2C_CMD, I2C1_STATE, I2C1_STATUS`
Timer 2	13	`TIMER2_CTRL, TIMER2_CMD, TIMER2_STATUS, TIMER2_TOP, TIMER2_CNT`
Pulse Counter 0 Pulse Counter 2	27 29	`PCNTn_CTRL, PCNTn_CMD, PCNTn_STATUS, PCNTn_CNT, PCNTn_TOP`

The registers listed in Table 22 are only registers which relate directly to interrupt generation by the peripheral—including the register which is used to enable the peripheral—and do not list other registers related to the general configuration of the peripheral (such as the timer's period or the pulse counter's overflow value).

Note that numerous other registers exist for each of these peripherals (the USB peripheral, for example, has a total of 83 registers). A full listing of registers for each peripheral is provided in the reference manual section for that peripheral.

Clock dependencies

The microcontroller on the development kit is intended for uses with a need for low energy consumption—in its second-lowest energy mode, it is rated to draw as little as 0.65 μA from the supply. In keeping with this, the microcontroller can individually enable and disable most peripherals and hardware devices. In order to use the peripherals (including those set out in chapter 14.2), the clocks for each peripheral must be enabled first. This section serves as a reference of the clocks required for each peripheral used in the firmware to function.

Table 23 lists the peripheral, the primary clock source, any secondary clock source, and the registers relevant in the configuration of those clocks.

Secondary clock sources are driven by primary clock sources, but may be divided to a slower clock rate. Each peripheral may have its own individual clock source driven by a primary or secondary clock source.

Table 23

Microcontroller peripheral clock dependencies

Peripheral	**Primary**	**Secondary**	**MCU Registers**
Timers 0–3	HFPER	—	`TIMERn_CTRL, CMU_CTRL, CMU_HFPERCLKDIV, CMU_HFPERCLKEN0`
Low-energy timer 0	LFRCO LFXO HFCORE ULFRCO	LFA	`CMU_LFCLKSEL, CMU_LFACLKEN0, CMU_LFAPRESC0`
ADC 0	HFPER	—	`ADC0_CTRL, CMU_CTRL, CMU_HFPERCLKDIV, CMU_HFPERCLKEN0`
Pulse Counter 0 Pulse Counter 2	LFRCO LFXO HFCORE ULFRCO	LFA	`CMU_PCNTCTRL, CMU_LFCLKSEL, CMU_LFACLKEN0, CMU_LFAPRESC0`
I^2C Bus 1	HFPER	—	`I2C1_CLKDIV, CMU_CTRL, CMU_HFPERCLKDIV, CMU_HFPERCLKEN0`
GPIO	HFPER	—	`CMU_CTRL, CMU_HFPERCLKDIV, CMU_HFPERCLKEN0`
LCD	LFRCO LFXO HFCORE ULFRCO	LFA	`CMU_LFCLKSEL, CMU_LFACLKEN0, CMU_LFAPRESC0`
	—	LFA_{LCD}[25]	`CMU_LCDCTRL`
DMA[26]	HFCORE	—	`CMU_HFCORECLKDIV, CMU_HFCORECLKEN0`
USB	HFCORE	—	`CMU_HFCORECLKDIV, CMU_HFCORECLKEN0`

Each primary source is driven by an oscillator. The available oscillators are listed in the microcontroller family reference manual, and are connected in a complex topology with each of the primary clocks (Silicon Labs, 2014a, p. 126). Oscillators

[25] The LCD framerate is the frequency produced by the further division of LFA_{LCDpre} (the clock divided from LFA and provided to the LCD controller). Its configuration registers are separate from those for the LCD controller clock.

[26] Direct memory access (DMA) allows peripherals to read from and write to the processor's RAM directly, without the processor's intervention.

are enabled and disabled using the `CMU_OSCENCMD` register.

Note that low-energy peripherals also require the `CMU_HFCORECLKEN0` register to have the `LE` bit set, as this clock is used for bus access (Silicon Labs, 2014a, p. 148).

Testing and verification

Appendix F6 contains a test plan which briefly sets out the considerations made in testing the proof-of-concept prototype and its firmware, and gives instructions for testing the prototype. The instructions are organised into discrete action items, and brief explanatory notes are provided for each action item.

Appendix F7 provides a description of all procedures performed in carrying out the action items, results of those procedures, observations made, and discussion related to the procedures and, where necessary, future actions.

15. Ancillary prototypes

The purpose of the proof-of-concept prototype, as noted in chapter 14, is to enable practical testing of two specific aspects of the controller. While designed with the testing of other aspects of the controller in mind—for example, by the inclusion of a temperature, current, and voltage transducers—connection of the proof-of-concept prototype to the development kit cannot occur until the safety of such a connection has been verified.

However, while connection cannot occur before safety is verified, it is also true that the development of other controller functionality cannot wait until safety is verified. In order to ensure that the functionality is completed in time, a number of ancillary prototypes—primarily consisting of software and firmware—are to be developed as discrete units. These units, while separate from the firmware which is to drive the proof-of-concept prototype, are to be developed so as to enable easy integration with that primary firmware. This separation also has the advantage of making easier the testing of the firmware and software by reducing the possible test surface for any one component.

These units, each an ancillary prototype, are to demonstrate:

- The viability of USB as a means of communications, the work necessary to implement USB communication on the sides of both the controller and the host computer, and the suitability of the protocol referenced in chapter 13.
- The use and methods of data acquisition, and the application of acquired data in the control of fans.

This chapter covers the design and development of each ancillary prototype.

15.1 USB prototype

One of the core features, and one of the distinguishing features, of the controller design is the use of USB to communicate with the host computer for configuration and status reporting. However, USB is not especially simple, and so it is necessary to split USB functionality into a separately testable, separately demonstrable unit. Any issues in the development of the USB demonstrator will then not impede the testing of other portions of the project.

As outlined in the introduction to this chapter, this prototype is to demonstrate the viability of using USB, what is necessary to implement USB, and the suitability of the protocol specified as part of the project. Potentially conspicuous by their absence, no requirements relating to fan control are present. This is intentional, as such requirements would not aid in demonstrating the use of USB or the use of the protocol.

General structure

The USB prototype is to consist of two main components—the firmware, present on the development kit and acting as the USB device; and the software, which is to run on a typical computer and provide a user interface to the device.

In a final design, the software would expose a number of settings configurable by the user, and would communicate those settings—using facilities provided by the operating system—over USB to the firmware. The firmware would then process the communication, validating it where appropriate, and act on the information as necessary. This action, for example, could be the adjustment of the speed of a fan, a request for current status data or sensor read-outs, or an instruction for the device to enter programming or firmware upgrade mode.

Requirements specification

The USB prototype is to consist of both firmware and software, and so a single program specification is not appropriate. While two program specifications could be provided, the simpler and more concise option is to provide an overarching requirements specification covering both firmware and software.

The USB prototype must consist of:

15.1.1 A software component which displays the fan status reported by the fan controller; which enables the user to instruct the fan controller to adjust the current state of the fan; and which does so in implementation of the device class specification contained in Appendix D.

15.1.2 A firmware component which implements the device class specification contained in Appendix D; and which displays a visual indication on each instruction by the software component (for example, by the display of text on an LCD, and without the need for the firmware to implement fan control routines).

15.1.3 A hardware component (including a processor on which the firmware is to execute) which is to be connected over USB to the host computer.

In addition, the firmware component specified in requirement 15.1.2 should be developed, as far as is practical, with regard to the aim of integrating the firmware component with the firmware produced for the proof-of-concept prototype.

Design and source code

Appendix G1 contains resources produced in designing the USB prototype, and discussion about relevant considerations made. Source code for the prototype is contained in Appendices C3 (for the firmware) and C4 (for the software).

Interrupt and clock dependencies

Refer to Table 22 and Table 23 in chapter 14.4 for information and discussion on general interrupt and clock dependencies.

15.2 Sensors prototype

In order to properly control a fan, the fan controller must have information about it, and that information must be gathered from sensors and transducers in the fan or the controller. The proof-of-concept prototype firmware largely focused on demonstrating the viability of methods of control of a fan, without including any functionality for responding to changes in environment. It is necessary, then, to demonstrate or consider that aspect of the fan controller.

Due to time constraints, it was not expected that there would be suitable time to design, produce, and test a sensors prototype, and so instead it will be considered what might be necessary to produce a sensors prototype.

This consideration is contained in Appendix G2.

16. Production design

The proof-of-concept and ancillary prototypes, while they provided insight into what would be necessary or desired in producing a final design, did not fulfil the aim of the project. The focus of these prototypes was on investigating or testing one or more specific aspects of the design of a fan controller, and not on what might be needed in a design for commercial or volume production.

To fulfil the aim of the project, this chapter will consider the information and the knowledge gained in implementing the prototypes in conjunction with what is considered best practice or what is required by standard or specification, and will hence make recommendations about a final design for a fan controller.

The first sections of this chapter are given in the same order as the chapters of the *Research and Theory* partition.

16.1 Computer fans

In implementing and testing the proof-of-concept prototype, it became apparent that changes were required to various aspects of the design. Further, as the proof-of-concept prototype did not fully meet the requirements in chapter 2, additional components must be included in the production design.

Speed control

As discussed in the section on fan speed monitoring below, use of the PWM DAC is unnecessary and provides minimal benefit over the direct switching of the fan supply. This means that there is no reason to include the PWM DAC in any final production design, and so speed control is greatly simplified.

Without the PWM DAC, all that is required for speed control in the production design is a suitably-rated transistor to control the supply to the fan, and a small transistor to pull the PWM control input to ground. The same type of transistor was used in the proof-of-concept prototype for each function, but the use of a smaller transistor in the production design will save some cost without any effect on the quality of the device.

The supply transistor must be rated for 12.6 volts and 3.3 amps drain-to-source, must have a gate-to-source threshold voltage suitable for operation with the 3.3

volts produced by microcontroller GPIOs, and must be capable of switching at a frequency of at least 25 kHz. The Fairchild FDMS7682 (2015a) used in the proof-of-concept prototype is a suitable MOSFET, and is available at reasonably low cost (as low as £0.31 in single units). This device is rated for current well above what is required—at the 15.2 mOhm $R_{DS(ON)}$ estimated in chapter 10.2 and taking maximum permissible power dissipation (given a maximum permissible increase in temperature of 75 °C) as 0.6 watts, a FDMS7682 could provide each fan with approximately 6.28 amps (discounting losses which result from switching). This additional current-carrying capability could enable the use of a single FDMS7682 to operate two fans at the same speed.

The PWM control input transistor must be rated to switch at least 5.25 volts, and must be rated for a current of at least 8 mA. The transistor should also be rated to switch at a minimum of 25 kHz, although 21 kHz is the minimum acceptable switching frequency. The proposed transistor is the Nexperia NX7002AK (2015), which is rated to switch 60 volts, for a current up to 300 mA, and to dissipate no more than 325 mW with $R_{DS(ON)}$ of 4.5 milliohms. When sinking 8 mA, this device will dissipate in the region of 0.29 mW, producing only a negligible change in the temperature of the device. Available at £0.11 in single units, this device enables a relatively significant saving compared to the FDMS7682.

For discussion relating to the determination of the speed control method for a particular fan, refer to the section on fan functionality monitoring below.

Monitoring: supply voltage and current

The testing of the proof-of-concept prototype did not indicate that the changing of the method of monitoring fan supply voltage and current was necessary. The methods used in the proof-of-concept prototype—a potential divider to measure the supply voltage, and a current-sense resistor for the current—are therefore proposed for use in the production design.

Monitoring: fan functionality

Although not included in the proof-of-concept prototype, it is necessary for the production design to include means of assessing the functionality of a connected fan. In particular, it must be possible to determine whether a fan is connected and whether it is a 2-, 3-, or 4-pin fan.

The connection state of a fan can be determined by energising it and observing the output from the current transducer. As current can only flow in a closed loop, there would only be a non-zero result from the transducer when there is a fan connected. This does not provide any notification immediately on connection, but this issue could be resolved by the fan controller periodically energising any known-unconnected fan connections to observe whether current flows.

To determine whether a fan is of the 2-, 3-, or 4-pin kind, the fan controller must rely on detecting features present only in a particular variety—in 3-pin fans, the presence of a tachometer; and in 4-pin fans, the presence of a control input. As such, 3-pin fans can be identified by energising the fan and observing whether any pulses are registered on already-present fan speed monitoring hardware. For the detection of 4-pin fans, there are two viable methods: first, the controller uses the speed control circuitry to provide a control signal and observes whether fan speed changes; or second, the controller detects the presence of the fan-provided pull-up on the control input.

Each of the methods for detecting 4-pin fans has its advantages: the first saves on additional circuitry, while the second enables simpler firmware and would permit faster initialisation by obviating the need for an adjust-speed-and-wait cycle. The additional circuitry required would be circuitry to detect a pull-up voltage in the range of around 2.5–5.25 volts, and so use of the control input voltage to drive a Nexperia NX7002AK transistor (see the section on fan speed control above) that is wired to an input on a microcontroller is likely to be the least expensive method using additional circuitry. Depending on the logic level thresholds for the device receiving the input, a simple potential divider may also be viable, if the divided voltages are within the safe operating range of the device and are appropriately above the logic high threshold voltage.

Any of these methods is viable, and so a final selection should be made once the characteristics of the receiving device are known and once it is known whether there is a need to cut cost.

Monitoring: fan speed

The proof-of-concept prototype enabled the testing of three methods of fan speed control: the varying of the absolute fan voltage supplied to a fan using a so-called PWM DAC; the varying of the average voltage supplied to a fan through the modulation of the supply by PWM; and the use of 4-pin fan PWM control. As discussed in chapter 5.4, it was anticipated that directly switching the supply would result in a highly distorted output from the fan tachometer, and so it was necessary both to have means of lowering the absolute voltage, and to establish what effect directly switching the supply would have on the tachometer output.

The testing carried out to establish this is covered in Appendix F7.7. This testing showed that, although the tachometer output is distorted as a result of the supply modulation, the distortion is not severe at or around 25 kHz. Given the relatively light distortion, it is expected that the signal could be made usable with minimal conditioning. Of additional note is the discovery in Appendix F7.9 that some fans may provide a tachometer where the output does not reach 0 volts—Fan 1A used in testing, for example, provided a tachometer signal varying between 2 volts and 12 volts. Such behaviour must be accounted for to ensure that the fan speed can be accurately recorded.

It is proposed that the signal be conditioned using a Schmitt trigger and a potential divider. The Schmitt trigger would be used to mitigate any effect of the voltage spikes observed in the tachometer output, with the potential divider to first lower the output voltage into the safe range for the trigger, and to second manipulate the tachometer output such that the spikes observed are do not cross between the triggering bands for the device. As the precise potential divider configuration depends on the operating characteristics of the trigger, a trigger must be selected before a suitable divider arrangement can be selected. The proposed selection is the ON NL37WZ17 (2013) triple Schmitt trigger non-inverting buffer, capable of 1.65–5.5 V operation and with parameters specified at 3 volts (approximately the output of the regulator integrated into the microcontroller on the proof-of-concept prototype). The NL37WZ17 has its positive-going threshold in the range 1.3–2.2 volts, its negative-going threshold in the range 0.6–1.5 volts, and its hysteresis voltage in the range 0.4–1.2 volts, with typical values of 1.9 volts, 1.0

volts, and 0.93 volts.

In selecting an appropriate potential divider for this device, it must be ensured that the positive level of the tachometer is at least 2.2 volts, that the negative level is at most 0.6 volts, and ideally that any noise does not spike more than 0.4 volts in any direction—any larger spike could pass from the trigger's hysteresis band into one of its trigger bands, even though the typical hysteresis band width is given as 0.93 volts. Additionally, it is necessary to consider the behaviour of a fan tachometer and the noise potentially introduced as a result of switching the supply. Reviewing Appendix F7.7, it was observed firstly that Fan 1A produced a tachometer signal which did not fall below approximately 2 volts, making a lower threshold of 3 volts is appropriate; and secondly that spikes in voltage were not greater than around 2.5 volts at the negative level and not greater than around 1.5 volts at the positive level, meaning that tolerance of spikes of magnitude 2.5 V is desirable. The potential divider must be selected so as to ensure the thresholds set out here line up appropriately with the thresholds for the trigger given above.

In order to determine a suitable division ratio, a simple spreadsheet of a column containing the tolerances in the preceding paragraph,[27] a column for the trigger thresholds, a computed column displaying the value of the first column multiplied by the division ratio, and a further computed column with a value that indicated whether the value in the first computed column was suitable. The ratio was then adjusted through coarse "basic" fractions—half, a third, a quarter, a fifth, and so on—until somewhat close divided values were obtained, and fine-tuned until the divided values were reasonably close to those desired. It did not appear possible to obtain all desired values; however, several potential ratios were identified and are set out in Table 24 below.

Table 24

Potential division ratios for conditioning fan tachometer output

Division Ratio			**Schmitt Trigger Input (Volts)**			
			Max.	**+Spike**	**−Spike**	**Min.**
		Target	>2.20	>1.80	<1.00	<0.60
$^{9}/_{44}$	20.4545%		2.33	1.82	1.13	0.61
$^{69}/_{352}$	19.6023%		2.23	1.74	1.08	0.59
$^{399}/_{2048}$	19.4824%		2.22	1.73	1.07	0.58

Note: The +Spike and −Spike voltages are the target voltages accounting for the desired tolerance for voltage spikes, and not the magnitudes of a spike in either direction.

As can be seen, no division ratio produces a set of output values where each value matches the desired value. Nevertheless, while these values do not match those desired, they are likely to remain suitable—if, in the spreadsheet, the tachometer

[27] For the avoidance of doubt, the tachometer output tolerances used were: for the positive maximum, 11.4 V (the minimum fan supply voltage); for the positive minimum, 8.9 V (the positive maximum minus the 2.5 V tolerance); for the negative minimum, 3 V; and for the negative maximum, 5.5 V (the negative minimum plus the 2.5 V tolerance).

output values are made more typical,[28] the negative spike voltage for ratio $^{9}/_{44}$ is the only value that does not match the desired values. This is also the behaviour observed when both output tolerances and trigger thresholds are made typical, and so $^{9}/_{44}$ is not a suitable ratio. The remaining ratios are suitable, and so can be used in selecting a suitable potential divider arrangement.

The selection of a suitable arrangement is relatively simple in concept—a pair of resistances (whether single resistors or a parallel resistor arrangement) must be selected which both matches the ratio and which can be made from the standard series of resistors set out in BS EN 60063 (BSI, 2015). Both 352 and 690 are values in the E192 series, although the lower E-number series (E96, E48, E24, etc.) tend to be more commonly available. The nearest E48-series values to these E192-series values are 348 and 681, equivalent to around $^{68.88}/_{352}$. Using the spreadsheet as a means of verification, this ratio was no better or worse, and so is suitable for use. Additionally, one online parts retailer listed 146 resistors of 68.1K and 86 of 348K (with respective prices as low as £0.07 and £0.26 in single units), indicating that there is unlikely to be any difficulty in sourcing resistors.

It is important to note that the tachometer output pull-up resistor will have some effect on the divider—a 4.7 kiloohm pull-up (as was used in the proof-of-concept prototype) gives the divider a total series resistance of 4.7 + 68.1 + 352 = 424.8 kiloohms, allowing a current of 28.25 μA to flow and so a voltage of 0.133 V to be dropped across the 4.7 kiloohm resistor. While this is no insignificant voltage, the spreadsheet indicated that, even with the worst-case tachometer output values adjusted to account for the dropped 0.133 volts, the divided output would remain as suitable for typical trigger thresholds. If the trigger thresholds are instead taken to be their worst-case values, the only change is that the positive spike tolerance may not be sufficient to ignore a negative-going spike of 2.5 volts. Considering that the maximum observed spike magnitude at the positive level was not greater than 1.5 volts, this is unlikely to cause an issue, and so there is no pressing need to adjust the 4.7 kiloohm pull-up value. If practical testing reveals that this drop is an issue, the drop could be reduced by increasing the resistances used in the divider and by decreasing the value of the pull-up.

16.2 Form factor

It was determined in chapter 7 that the fan controller should be mountable in a typical 5.25" bay, and that the reference should be SFF-8551J (which is concerned with the form factor of CD drives). This remains the recommended form factor for a production design, and so the use and implementation of this specification is considered in this chapter.

Physical dimensions: review

The specification sets out recommended dimensions in Table 5-1 and its related figure 5-1 (SFF, 2000, pp. 9–10). These dimensions are specified as ±0.25 mm (or equivalently ±0.010"). The general dimensions are a width of 146.05 mm (5.750"), a length of 202.80 mm (7.984"), and a height of 41.53 mm (1.635"), with the length and height noted as maximums. The remaining dimensions give information such

[28] The tachometer output values are made more typical by adjusting the positive maximum to 12 volts, the positive minimum to 9.5 volts, the negative minimum to 2.5 volts, and the negative maximum to 5 volts.

as screw locations and bezel thickness.

While it would be possible to have manufactured a circuit board that would fit these dimensions, it would likely be prohibitively expensive—using as a guide the circuit board manufactured for the proof-of-concept prototype, each board of area 18.26in^2 would cost $5.94 in 500-unit volume, around $0.325/in^2, resulting in a circuit board of dimensions 5.75"×7.984" likely costing $14.93, or £10.71 at the exchange rates at the time of writing. This represents more than a quarter of the target per-unit cost in requirement 2.1.8. Instead, the production design should use only as large a board as necessary, which should be fixed to a metal sled which interfaces to the bay and its screw locations.

Metal sled design

The proposed design for the metal sled is simple—a U-shaped construction, with the fan controller screwed to base of the inside of the U and the sides extending only as far as necessary to enable attachment to the computer chassis. The design would not include a bezel for the front of the computer chassis, and would have the fan controller attached such that the fan connections were easily accessible from the inside of the chassis (such as at the extent of the permitted length for a 5.25" CD drive, around 7.984" or 202.80 mm from the front of the chassis.

A metal is recommended for the material to provide a path to chassis ground. The choice of a particular metal is likely to depend on the required cost-effectiveness, although prior knowledge of typical computer chassis construction would suggest use of steel or aluminium (these being common materials from which to build a computer chassis). This is partially confirmed by a review of the chassis listed in Table 2—of the chassis listed in that table, five were steel while the material for one was not specified by its manufacturer. Aluminium is more common in higher-cost chassis, and was present in 2 of 4 randomly-reviewed chassis over £150.

16.3 Power delivery

The implementing of the prototypes did not raise any concerns with the method of power delivery, and so the hybrid PCI-E 12 V and USB 5 V system remains the recommended system. This chapter considers the implementation of the power delivery system in a production design.

PCI-E 12 V supply

On the proof-of-concept prototype, a 12 V supply was connected using a screw terminal so as to enable easy connection to a bench power supply. For a final, production design of the fan controller, this must be replaced by a connector that is capable of accepting PCI-E 2 × 3 auxiliary power connectors. Such connectors are available from Molex, in two varieties: 45718, which is a straight header rated for 13 amps per contact; and 45558, which is a right-angle header rated at 8 amps per contact. A third part, 45732, is listed on the Molex website, but appears to be identical to 45558. The right-angle variant of this connector is likely to be the most convenient for a user, as vertical access to a fan controller mounted in a 5.25" drive bay is likely to be hindered by optical disc drives (or other equipment) mounted in adjacent bays.

If the fan controller were to be produced in a higher power variant, Molex does also appear to manufacture connectors compatible with PCI-E 2 × 4 auxiliary power connectors (part 45586), but also appears to have removed product pages for this and related connectors from its website. A number of online retailers list the item for sale, albeit generally as a non-stocked item.

USB 5 V supply

The proof-of-concept prototype did not include a 5 V supply—the connection to a USB host was through the microcontroller development kit, which was only connected to the prototype where absolutely necessary. Instead, a 3.3 V supply was provided from a bench supply using a connection on the prototype's 20-pin female pin header. This is clearly unsuitable for a production design, and so there must be a suitable USB connector.

This would, at first, appear simple—a number of standard USB connectors exist, almost all of which are easy and inexpensive to source. However, a standard USB connector is not typically available internally, and so a pass-through cable (which would be routed from a port internal to the chassis to an external port, such as might be available from the chassis back panel) would be necessary. Further, as chapter 9.4 noted, there exists a standardised internal USB connector.[29] This is a connector more suited for internal use (that being its intended purpose), but may not be a connector whose availability is guaranteed—although no motherboard given in Table 8 had fewer than three internal connections, a brief survey of the stock of an online retailer indicated than many chassis include four front panel USB ports, and so (with $^2/_7$ of those motherboards having four or fewer internal connections) it is necessary to consider other means of connection.

While the "sledgehammer solution" of having both an internal USB header and a standard USB header exists, including an adaptor cable is the most suitable choice of solution. An internal-to-internal cable would be required regardless, and so it could be possible to make an agreement with a supplier for both. As an example, StarTech already produces both internal-to-internal (2018a) and internal-to-USB female (2018b) adaptor cables. Although the second cable is not suitable for this use (and so would be replaced by a bespoke cable), it would not be unreasonable to anticipate the total cost of two cables (given the single-unit prices of £2.39 and £3.59, respectively) to be less than £2 in volume. No USB male-to-internal cables could be identified other than non-name-brand cables from private sellers.

Protection and control: general

In order to help ensure that the fan controller is safe for use, circuit protection and control devices must be included. However, the approach differs for each of the supplies in use, and so each requires specific consideration.

Both supplies require protection against overcurrent and overload current. This must include protective devices which can operate independent of a processing element (such as a microcontroller).

For the 12 V supply, there must be a method of isolating the entire power network from the supply. It must also be possible to individually energise or de-energise each fan connected to the controller, and there must be protection against the

[29] It is important to note that the standardised internal connector is not part of the USB standard, instead being part of a specification published by Intel (2005c) which provides a set of "connection and mechanical recommendations for all main boards having internal connectors requiring external connection." Here, USB ports on the front of chassis.

inductive flyback produced by each fan.

For the 5 V supply, as a result of the design of the internal header, there must be protection against the swapping of USB V_{BUS} and GND conductors.[30] V_{BUS} and the USB data lines must also be protected against transient overvoltage.

Protection and control: 12 V supply

The proposed overcurrent and overload current protection for the 12 V supply is the combination of a single non-resettable fuse and one resettable "polyfuse" for each fan. The intention is for each resettable fuse to trip above the maximum fan steady-state current of 1.5 A (with a tripping characteristic enabling operation for brief periods at the maximum start-up current of 2.2 A), providing a basic layer of overcurrent and overload current protection. The non-resettable fuse would be rated to trip below the maximum fault current[31] of the selected resettable fuse, ensuring safe operation under short circuit conditions.[32]

Supplementary overload current protection would be provided by the controller, which would actively monitor the output of current transducers and would cause portions of the circuit to be disconnected or de-energised as appropriate.

It is recommended that inductive flyback be protected against using a diode rated to carry 3.3 A at 12.6 V. One such diode should be provided for each fan.

The controls enabling this would also be a combination—a relay (whether solid-state or electromechanical) placed to enable complete isolation of the controller from the supply, and a MOSFET as a low-side switch for each fan. The relay used must be selected with an especial focus on low control (or coil) current, as even a current of 50 mA would represent a significant proportion of the 500 mA total available from the USB connection. Irrespective of the precise control current, it would be necessary to control the relay with a small transistor—a microcontroller generally cannot source more than a handful of milliamps, and even the relatively high 20 mA sourcing capability of the controller used with the proof-of-concept prototype would likely be unable to control the majority of relays. Alternatively,

[30] To elaborate, while the internal USB header includes a keyed fifth position, this does not prevent the connection of a 4-pin plug (which, without any key, could be inserted either in the correct or incorrect orientation). An example of a 4-pin plug is the plug included with the StarTech USBMBADAPT USB Type-A to internal header adaptor cable (2018b).

No protection is required for the connection of V_{BUS} or GND to a data line, as section 7.1.1 of the USB specification (USB-IF, 2000, p. 124) requires that a USB transceiver be capable of withstanding a continuous short circuit between the a data line and V_{BUS}, GND, the other data line, or the cable shield for at least 24 hours at the maximum V_{BUS} of 5.25 volts.

[31] "Maximum fault current" for resettable fuses is the equivalent of breaking capacity for typical fuses and circuit breakers; a non-resettable fuse does not break the circuit (instead introducing a large resistance to limit current), and so differing terminology is used.

[32] Prospective short-circuit current is estimated at 35 amps. The ATX12V specification recommends use of 16 AWG wire (Intel Corporation, 2005b, p. 37), which has a standard resistivity of 10.36 mOhms/metre for 100% IACS-conductivity copper (ASTM, 2002). Using a conductor length of 35 cm in each direction (the shortest length specified for a power supply in Table 3, for the be quiet! BN240), this gives a minimum total-circuit resistance of 2×3.62... mOhm = 7.25 mOhms. However, the terminations, thin circuit board traces, and the $R_{DS(ON)}$ from transistors likely add resistance in the region of 150–350 mOhm, which would produce a prospective short-circuit current of 35–80 amps. This calculation did not consider any correction factors.

if a suitably high-current device could be found, the relay could be controlled by a digital or opto-isolator. This would remove current pressure on the 5 V supply, but is likely to be a relatively expensive option. It would be necessary to compare projected current demand from all 5 V-powered devices against the 500 mA that the USB connection can provide, which could only realistically be done after or as part of component selection.

Regarding the control MOSFETs, the only particular requirement is a low drain-to-source on resistance—with the requirement to be able to switch 3.3 A, as in the proof-of-concept prototype, a MOSFET with a 2 ohm $R_{DS(ON)}$ would dissipate in the region of 22 W. No reasons preventing or discouraging use of the Fairchild FDMS7682 MOSFETs used in the proof-of-concept prototype were identified.

Protection and control: 5 V supply

The proposed method of protecting against overcurrent and overload current in the 5 V supply is the use of a single resettable fuse on the USB V_{BUS} line. The use of a non-resettable fuse for overcurrent protection is not considered necessary as a result of the standard short circuit behaviour (see footnote 30 on page 76). A resettable fuse selected for this purpose should operate above 500 mA, or lower if a more accurate estimate of current demand is available.

Transient overvoltage protection can be provided by an array of TVS diodes on the V_{BUS} and data lines. Alternatively, more specialised devices intended for use in USB protection are available—for example, a Silicon Labs application note on the design of USB hardware (2013c) references the Nexperia IP4220CZ6 (2011a), a monolithic device incorporating a TVS diode array and a Zener diode and which is intended specifically for use in the protection of USB interfaces.

Protection against the swapping of V_{BUS} and GND is slightly more complex—this requires a device to permit current in only a single direction, and so a diode on V_{BUS} would initially appear to be the obvious solution. However, the forward voltage of a silicon diode of around 0.7 V would pull V_{BUS} from any valid value to a value significantly below the 4.75–5.25 V permitted. Even the markedly lower forward voltage of Schottky diodes, at a value likely not less than 0.2 V, would be too great a drop, pulling any voltage less than 4.95 V below the limit. The solution is to use a so-called “ideal” diode—a MOSFET, controlled in such a way as to act as a diode, where the forward voltage is a function of the on resistance and the current through the device (and so can be in the tens of millivolts range).[33] These devices are available in both monolithic (with MOSFET) and controller (using an external MOSFET) types.

In this application, a device such as the Maxim MAX40200 (2017) is a suitable choice. The device is available at reasonable cost (£0.47 for the WLP package in single units), provides a low forward voltage of 43 mV typical at 500 mA, operates from a 1.5–5.5 V supply, and is rated to block up to 6 V (safely above the 5.25 V maximum voltage on V_{BUS}). It is proposed that the production design use two of these diodes (one for each of V_{BUS} and GND) for two reasons: firstly, a path from V_{BUS} to ground may exist even with USB GND disconnected (for example, via the 12 V supply); and secondly, this causes USB GND potential to be around 43 mV

[33] Note that variation in nomenclature exists. “Ideal diode” appears to be the most common term, although some sources use “smart diode,” “lossless diode,” or “active diode” instead.

above circuit ground, which cancels out the forward voltage drop from the diode placed on V_{BUS}. While this technique would work with conventional diodes, heat loss in those diodes would remain unsuitably high: at 500 mA, 350 mW for a 0.7 V silicon diode and 100 mW for a 0.2 V Schottky is enough to cause typical surface-mount devices to experience increases in temperature up to 60 °C and 100 °C, respectively (Fairchild, 2015b; Central, 2013). At the 75 °C ambient assumed for the proof-of-concept prototype, this would bring the silicon device within 15 °C of its maximum operating temperature, and would cause the Schottky to exceed its maximum operating temperature. In comparison, the 43 mV dropped at 500 mA is equivalent to 21.5 mW, and would produce a rise in temperature of 2.3 °C.

16.4 Host–controller interface

The selection in chapter 9 of USB as the host–controller interface remains valid, and nothing learned in implementing the prototypes indicated that there would be any advantage in using another interface. However, while USB remains the most appropriate interface, USB 2.0 is not the most appropriate revision.

As discovered whilst implementing the USB prototype (see Appendix G1), it is not possible for a USB device to request that the Windows operating system load the WinUSB driver stack without implementing portions of USB 3.0. In order to have the WinUSB driver automatically loaded, then, the fan controller must identify itself as supporting "USB 2.1" (i.e. USB 3.0 operating at USB 2.0 speeds), and must respond to standard USB requests for BOS descriptors.

While it would be possible to have WinUSB loaded for a USB 2.0 device, this may require a more complex device set-up process—a driver package would become necessary, with a custom-written configuration file specifying that WinUSB is to be installed for the device (Microsoft Corporation, 2017e). It may also be the case that this point is moot—if support for operating systems older than Windows 8 is desired, support for the automatic loading of WinUSB would not be universal, and so use of a driver package would be necessary regardless. Additionally, there may be more advanced features (such as using Windows Update as a means of distributing firmware updates) which require use of a driver package. However, whether this is or is not the case, there is value in supporting use of the BOS, as it may enable use of particular features under other operating systems.

Refer to chapter 16.3 for discussion relating to the physical interface.

16.5 Driver stack

As discussed in chapter 11, WinUSB was selected as a base for the host computer driver as it enabled simplified development. This remains a sensible choice, and no pressing need to select a different architecture was identified.

For discussion on the fan controller protocol, see chapter 16.6. Appendices G1.4 and G1.5 cover the development of software for use with the USB prototype (see chapter 15.1), and so contain relevant information and discussion.

Windows.Devices.Usb

Although WinUSB was the underlying driver stack in use, the driver produced for the USB ancillary prototype used the `Windows.Devices.Usb` abstraction to ease development. It is recommended that, in a production design, this abstraction not be used, as several important limitations make it unsuitable for production use.

First of these limitations, the abstraction provided no useful means of handling a transfer error. The `SendControlInTransferAsync` method (and the equivalent method for control OUT transfers) return an `IAsyncOperation` object, which can indicate four basic statuses (started, cancelled, completed, or error) and can provide an `HRESULT` error code. While this could be used to handle errors, there is no list of typical error codes returned, and the list of standard Windows error codes contains thousands of codes. It could be assumed that the errors returned by the abstraction are the same as might be returned by WinUSB directly, but, as a note to this effect could not be found in documentation, it cannot definitively be said that this is the case. Additionally, it is unclear whether errors specific to the abstraction can be raised, or if all errors raised would be WinUSB errors.

Second, the abstraction is designed for the Universal Windows Platform, and so uses objects specific to UWP. If the driver targeted the .NET Framework (as was the case for the USB prototype), this would require the use of a number of conversion utilities wherever UWP-specific objects were used. Further, as UWP can only be used with Windows 10, the abstraction could not be used for versions of the host computer driver targeting older operating systems (such as Windows 7 and Windows 8 or 8.1). As Windows 7 is used by 41–48% of Windows users (with 39–44% using Windows 10), use of the UWP abstraction would require either that all non-Windows 10 users be disregarded, or that separate drivers only for older versions of Windows be developed (NetApplications.com, 2018; StatCounter, 2018). Neither is a good option—the first could half the number of potential users, while the second would add significantly to the development required, especially considering that separate (or largely separate) drivers would already be necessary to support Linux-based or macOS operating systems.

Third, the abstraction does not expose the full capabilities of WinUSB. This is not a great issue—some of the features not exposed were not considered for use in the protocol implemented by the USB prototype—but the availability of features not available in the abstraction could simplify code. Briefly, the abstraction does not appear to support use of isochronous USB transfers, power management, WinUSB pipe policy control, or the ability to reset a USB device or use WinUSB functions synchronously. Of these unsupported features, the ability to execute a WinUSB function synchronously would immediately enable the simplification of code, as the boilerplate required with asynchronous execution could be removed from parts of the code where asynchronicity gives no benefit.

These limitations in mind, it is recommended that the production design not use the `Windows.Devices.Usb` abstraction.

Alternatives to Windows.Devices.Usb

It being recommended that the `Windows.Devices.Usb` abstraction not be used, potential alternatives must be considered. A number of viable options exist.

The simplest option available is to continue to use WinUSB, either directly or via another abstraction (whether bespoke or already available). One abstraction is the open-source libusb (Dickens, 2017), which includes support for WinUSB as well as the comparable interfaces on a number of other platforms. As stated at the start of this chapter, no pressing need to change from WinUSB was identified, and so libusb is likely to be the best choice.

On Windows, an interface to USB devices is exposed through the User Mode Driver Framework (UMDF). However, as stated in documentation for UMDF, this interface is an abstraction of WinUSB (Microsoft Corporation, 2017d). Use of this framework would therefore provide no advantage over use of libusb.

If a specific need for the access provided by that type of driver existed, it would also be viable to use a kernel-mode driver. No situation necessitating the use of a kernel-mode driver for the fan controller is foreseen. In the context of USB, it would be necessary to use a kernel-mode driver if the ability to change the USB device configuration[34] is required (as WinUSB does not expose this functionality), but no normal situation where this would be a requirement is foreseen.

General architecture

The least complex choice of architecture is that used in the drivers produced for the USB prototype: a simple utility, manually started by a user and which provides a simple interface for device configuration and status monitoring. The controller would, under this architecture, operate autonomously in the configuration it was last placed in until next reconfigured. This option is viable but not preferable, as it provides no real advantage or unique selling point over other controllers.

Instead, it is proposed that the driver operate as a service (or daemon), constantly running in the background on the host computer. This architecture has a number of advantages—first, the host computer can more easily monitor the controller's status and inform the user of any relevant changes in status; second, the service could (with a suitable USB protocol) receive and relay data to the fan controller, enabling the use of almost any data (such as processor utilisation) in controlling the fans; and third, a service architecture would reduce client-side complexity in a transition from a WinUSB-based driver to another driver base, as the underlying driver implementation is not exposed to the client (enabling use of an unchanging interface to the driver, and so an unchanging client). Additionally, there would be no practical restriction on the service being capable of operating in the configure-and-forget mode discussed in the previous paragraph.

In using service architecture, it should be considered whether an out-of-the-box interface capability should be included. For example, a scripting interface (using a language such as Tcl or Lua) would enable a user to write a simple, plain-text script to instruct the fan controller without the need for additional software. This would add considerably to driver complexity, but could be added after any initial release or publication.

16.6 Protocol

The protocol sets out how the host computer and the fan controller communicate over the host–controller interface. The protocol for the USB prototype is set out in Appendix D, with the implementation detailed in Appendix G1. However, the implementing of the protocol highlighted the unsuitability of that protocol for use in a production design. As such, it is recommended that the Appendix D protocol not be used, and that a new protocol be defined for a production design.

This chapter will consider the design of that new protocol.

[34] "Configuration" here meaning a grouping of interfaces exposed by a USB device, one of which may be active at any time (USB-IF, 2000, p. 244).

Flaws in the Appendix D protocol

The Appendix D protocol has numerous flaws—the protocol is overly complex for the little it does, and would not lend itself to future extension; it included no facility for retrieving an indicator of the last error; and the request for retrieving the current fan configuration, `GET_FAN_MODE`, is a crude method of retrieving the current fan configuration. Additionally, the protocol was underspecified in areas where precision is important (such as in the precise format of the configuration data provided to the controller).

Any new protocol design must seek to remedy these flaws, and should aim to be simple, extensible, well-specified, and efficient.

Architectural changes in the new protocol

In remedying the flaws in the Appendix D protocol, it is necessary to make some changes to the general architecture of the protocol. First of these changes, it is recommended that a new protocol be specified for USB 3.0 and not USB 2.0. This would enable use of the Microsoft OS Descriptors 2.0 functionality (and so, for example, the automatic loading of WinUSB) and the use of any similar features for other operating systems.

The second of these changes relates to the general flow of data between the host and the controller. In the Appendix D protocol, the host expressly requested that the controller provide some information, with separate USB requests for each item of information. This is not particularly efficient—a host must track intervals for requests (as appropriate), and must be provided with (and must process) any requested data, whether or not the values have changed. Further, this approach makes a full status update more difficult than is necessary—for example, if there were five separate items of information comprising a full status report, this might result in a host being required to make five separate requests (with all necessary validation and error-handling for each of the individual requests). Instead, it is recommended that a new protocol use two endpoints—the first being the default control endpoint required by the USB specification, and the second being an interrupt endpoint specific to the fan controller protocol.

In this configuration, the host would periodically request a status update from the controller on its interrupt endpoint, and the controller would provide a partial status update containing only the information which had changed since the status was previously requested. To make simpler the decoding of a partial update, the protocol should be specified in such a way as to enable the lengths of the fields of the status update to be determined with as little pre-decoding of the data as possible. The proposed solution is the use of a response where the first byte (or bytes) are a bitfield, with a set bit indicating that the information represented by that bit is present, and the fields being present in their order in the bitfield. Each field would then be a constant width (or a constant width multiplied by a factor which could be determined ahead of time, such as the number of fans) to enable the pre-allocation of buffers and the further simplification of the protocol. As for extensibility, this could be achieved by reserving one of the bits of the bitfield as an indicator that a further bitfield follows. Following this scheme, a host could successfully interpret the contents of a status update without prior knowledge of content it does not support—the standard encoding for variable-length bitfields would enable the host to walk through the data until it is indicated that no further bitfields are present; and the set ordering of the status fields would mean that any newer fields would appear at the end of the data, allowing a host to simply read up to the field it supports. Care should be taken in specifying the status fields if any fields are to be optional—a scheme such as that described in this paragraph would not enable the cherry-picking of features, as the decoding a future field requires knowledge of the lengths of all prior fields.

Such a scheme is likely to reduce the number of USB requests required for a fan controller to operate, although it would be necessary to retain a number of basic requests—in particular, requests for setting and retrieving fan configurations, and a request for instructing the controller to provide a full (rather than partial) status update on its interrupt endpoint. This latter request is necessary to solve the issue of a host driver having only partial updates without a status base to update.

Specific changes in the new protocol

In addition to the general architectural changes which are recommended for any new protocol, the implementing of the Appendix D protocol made clear a small number of specific areas where changes were necessary or preferable.

First, a new protocol should enable error handling—the basic statuses provided by USB (success, failure, etc.) would not be sufficient to diagnose an error, and so the protocol should allow a host to retrieve an error code. Retrieval need not be complex—for example, a USB request to retrieve a simple numeric error code for the last error raised would be simple to implement and use. It may be desirable to investigate other methods of error reporting, but these are likely unnecessary.

Second, in line with the aim of protocol simplification, unchanging information about the controller and its fans should be condensed into a single class-specific descriptor. In particular, information such as a unique identifier for each fan, the methods of speed control supported for a particular fan, and the information in the Appendix D fan controller configuration descriptor should be condensed into one descriptor. In order to enable future extension, it may be suitable to specify that any fan information field has variable length. The length, specified as part of the descriptor, would enable the host to either interpret the field (using its length to determine its content) or to ignore any portions of the field it does not support. As for the extensibility of the descriptor itself, the appending of fields to the end of the descriptor is likely to be suitable—the USB specification states that "if a descriptor returns with a value in its length field that is greater than defined by this specification, the extra bytes are ignored" (USB-IF, 2013, § 9.5). While this requirement applies only in relation to standard USB descriptors, its extension to class- or vendor-specific descriptors is unlikely to astonish.

Third, it is recommended that the setting of a fan mode be simplified through the definition of a separate request for each mode. Although this would increase the number of requests, this change would enable code simplification (with each of the request makers and request handlers requiring less logic) and would simplify future extension of the protocol (as the maximum permitted number of requests is much greater than the number of modes that the 2-bit field in the Appendix D protocol could specify, and because the behaviour for handling an unrecognised request is already standardised in the USB specification).

Fourth, the rules on the format of configuration data should be made stricter and more precise. In particular, where that data specifies a number of setpoints which are selected based on temperature, the protocol should require that the setpoints be ordered by temperature (and should impose similar requirements where the selection is based on another quantity). Additionally, the protocol should provide for a limit on the number of setpoints a host specifies (whether using a set number or by enabling a fan controller to specify its supported maximum). The protocol should also be extended such that the range of valid values for configuration data includes all (or most) values a controller could reasonably encounter.

Fifth, requirements relating to the characterisation of a fan by a controller should be extended. The protocol should require that a controller report the minimums for each of the modes of control it supports (minimum starting voltage for voltage control, minimum speed in RPM for absolute speed control, minimum percentage speed for percentage-based speed control, and so on). It may be suitable for this information to be included in the condensed descriptor discussed in the second specific change discussed above, whether as part of the discussed variable-length fan information field or as a separate field. Additionally, the representable range of values for these minimums should be expanded—the 0–31% range used for the minimum percentage speed in the Appendix D protocol is only suitable for fans which comply with the 4-pin fan specification, and so the upper 30% limit given in that specification may be exceeded by fans of a different specification.

Sixth, the protocol should enable the control of fans using measurements taken by the host computer drivers (as discussed in chapter 16.5, where fan speed might be set based on processor utilisation or another quantity measurable by the driver and not the controller). This control should be generic, and could take a number of forms in implementation—the simplest form would have the drivers perform all work and instruct the controller to set a fan for a single specific speed, while a more complex implementation could have the driver provide a function (similar to a speed-against-temperature function) and the measurements to be used as the inputs to that function. While the first option would greatly simplify the feature, the second option could have some minor advantages (such as the ability to set a default start-up value). Ultimately, the choice of implementation should be made as part of the process of fully specifying a new protocol.

16.7 Miscellaneous

The previous sections of chapter 16 made recommendations largely stemming from knowledge and experience gained in implementing the proof-of-concept and ancillary prototypes. However, the implementing of the prototypes does not cover all relevant areas, and so discussed here are miscellaneous areas of interest which did not fit in previous sections.

Firmware storage and upgradeability

In order to fulfil requirements 2.2.4 and 2.3.3, a production design for a controller must provide means to update the controller's firmware. As chapters 9.4 notes, a standard method of accomplishing this is defined for USB devices in the Device Firmware Upgrade (DFU) device class specification (USB-IF, 2004).

A device that supports DFU indicates its support by presenting to the host a USB interface descriptor with the device class information set to the standard values for DFU. Then, to initiate an upgrade, the host issues a `DFU_DETACH` request to that interface and resets the device. After this reset, the device reports to the host only the appropriate DFU descriptors (which the host uses as part of the upgrade process in assessing how the device will behave) and waits for the host to begin transferring the device the upgrade. Once this transfer completes, the host again issues a reset to the device, causing it to return to normal operation. This process is straightforward, but its implementation is complicated slightly by limitations in WinUSB—the driver stack does not expose any interface for the issuing of a reset to a USB device. To ensure correct function, either the host computer driver must request that the operating system prepare the device for safe removal using the `CM_Request_Device_Eject` function, or the fan controller be able to generate a detach–attach cycle when a `DFU_DETACH` request is issued. As there are likely to be similar limitations with other operating systems, the latter option is preferable.

As for storing the firmware, the most preferable option is some form of removable storage attached to the controller. This would have a number of advantages—the storage could be easily replaced if it fails, the effect of a firmware bug preventing use of DFU is reduced (as the new firmware could be placed on the removable storage device manually if required), and a user could be more easily permitted to load unofficial firmware onto the device, and this would enable the use of the controller by a hobbyist for a purpose other than to control computer fans. The most suitable removal storage format is likely Secure Digital (SD)—such cards are widely available, relatively inexpensive, and can be interacted with using the SPI bus commonly included with microcontrollers (SD Association, 2017, p. 209). The SD Association states (2018) that a licence is required to produce a compliant SD product; however, this licence appears only to cover use of SD Card trademarks, full SD Card specifications, and patents relating to the design and construction of memory cards, and so there is unlikely to be any need for a licence in developing the fan controller—there would be no need to advertise SD card support, and so no need to license SD Card trademarks; the controller would be developed using the SPI bus (rather than any other SD-specific bus) and the publicly-available SD Simplified Specifications (or, if these are encumbered by a licensing requirement, other publicly-available resources), and so there would be no need to license the use of the full SD Card specifications; and memory cards used in the controller would be sourced from a manufacturer, and so a licence for the patents required to manufacture such a card would not be needed.

In terms of cost, it is expected that including an SD card and socket would be relatively expensive—sockets for microSD cards are available starting at £0.80 (hinge type, Molex 47219-2001) or £0.70 (push-pull type, Molex 47571-0001) in single units, and with consumer retailers offering 8 GB microSD cards for prices in the region of £4.40, a lower-capacity card bought in bulk directly from the manufacturer or from a distributor is expected to cost in the region of £2. Using the £40 overall cost target given in requirement 2.1.8, £2.80 would represent around 7% of the total cost. A decision on the inclusion of removal storage should therefore be taken after the cost of essential components is known.

If removable storage is not included, it would be necessary to include some form of memory in the fan controller circuit. While microcontroller on-board memory could be used for this purpose, this would not be ideal—the EFM32WG990 uses flash storage, and so is not well suited to small writes.[35] Instead, a storage device supporting byte-level erase operations and with sufficient capacity only to store configuration data should be used. Such devices are available at low cost—the ST M24C16 (2017) is a 2048-byte, 400 kHz I^{2}C-connected EEPROM with a rating of 4 million write cycles available from £0.14 in single units, with smaller memories available at marginally lower prices. If greater endurance is required, the Fujitsu MB85RS16N (2015) is a 2048-byte SPI-bus ferroelectric RAM (FRAM) rated for ten billion write cycles and one trillion read cycles, and can be found at £0.85 in single-unit quantities.

Microcontroller-autonomous fan monitoring, etc.

As the requirements for the fan controller stand, there is unlikely to be a scarcity of PWM generators—the EFM32WG microcontroller in use includes four timers with three PWM channels per timer (and with independent duty control for each channel). Even if two timers were unusable in generation of PWM signals (if, for example, they were used as periodic timers in the firmware), this would leave six PWM channels with independent duty control. Each of those channels can additionally be routed to up to six separate pins on the microcontroller, a feature which (given that modulating the supply and providing a standard control signal would never occur at the same time for a given fan) could enable a single channel to be used in controlling a fan instead of the two that might initially seem necessary. However, if the requirements are changed such that additional PWM generators are required, it may be necessary to investigate other solutions.

One potential solution would be the use of a field-programmable gate array—a device consisting of hundreds or thousands of look-up tables (LUTs) representing particular logical expressions. Such a device could be used to synthesise as many PWM generators as are required (limited by the LUTs available), and so could significantly reduce any pressure on timer and PWM resources. Further, such a device could (with sufficient LUTs) implement other functionality, such as pulse-counting to determine fan speed or an interface to an SD card or other storage (if included as the section on firmware storage and upgradeability above discusses). Such a solution, while it would increase the cost and complexity of the overall design, could enable simplified circuit design—an FPGA that has an I^{2}C interface could be placed relatively close to controlled fans, with only connections for I^{2}C required to be routed back to the microcontroller. The Lattice iCE40UL1K (2016)

[35] Flash memory and EEPROM (electrically erasable programmable read-only memory) use comparable technology, with the primary difference being that flash memory is erased in portions of a fixed number of bytes (called pages) while the individual bytes of an EEPROM can be erased. The result of this is that, with flash memory, a change of even a single byte results in the erasure of an entire page (which may be, for example, 512 bytes). Further, as it is the erase (or rewrite) operation which degrades the cells of these devices, the repeated writing of a small number of bytes to flash is, proportionally, more damaging than writing a small number of bytes to EEPROM (Silicon Labs, 2013a).

is a small FPGA with 1248 LUT4s,[36] 56 kBit of RAM, two I^2C cores, and up to 26 programmable input–outputs, available from £1.16 in single units, and so would be a likely candidate if an FPGA were to be used.

Additional analogue-to-digital converter channels

As chapter 14.1 (in the sections relating to supply-monitoring transducers and the temperature sensor) notes, the availability of analogue-to-digital converter (ADC) channels is a limiting factor in the design. The monitoring of each fan requires at least two ADC channels (one for supply voltage, and another for current), and so the eight channels provided by the microcontroller used in the proof-of-concept prototype are entirely consumed in a fan controller design supporting the control of four fans. If it is desired in a production fan controller design to monitor more than four fans, or to monitor other quantities (such as overall supply current), the number of available ADC channels must be increased.

The simplest method of increasing the number of available ADC channels is to include in the design an external ADC, either to be used in combination with the ADC on the microcontroller or to enable the use of a lower-end microcontroller without an ADC. The Maxim MAX11643 (2011), for example, provides 16 channels with 8-bit resolution and a 2.5 V internal reference, and can be found at £2.11 in single quantities. While this would be an expensive option, especially considering that volume discounts did not appear to change past 100-unit orders, a device of this kind would triple the number of available ADC channels and could reduce the complexity of a circuit board (only requiring the routing of a serial connection between the microcontroller and analogue sources, rather than requiring that all analogue sources have a connection routed to the microcontroller).

There does, however, exist a less expensive alternative—both the 4000 and 7400 series of logic chips include a number of analogue switches. The connection of a single-pole many-throw switch (i.e. a multiplexer) would enable each channel on the microcontroller to act as many channels, effectively increasing the number of channels available for use. As with the solution given in the above paragraph, this solution would enable the circuit board simplification (as only a single analogue signal would be routed to the microcontroller for each switch, irrespective of the number of throws the switch has). In selecting a particular switch, it is preferable to have knowledge of the quantities to be measured—a downside of an analogue switch being used in this way is that the microcontroller ADC's scan mode cannot scan through all inputs of the switch, and so there is a trade-off between ability to scan through the inputs (by having more switches each with fewer throws) and the complexity of routing and firmware (with fewer switches reducing required control logic and reducing the number of connections to be routed). In terms of cost, either option (many switches or many throws) is cheaper than the use of an external ADC—both the Nexperia HEF4053B (2016a) triple SPDT and the Texas

[36] A look-up table (LUT) in an FPGA is generally specified in terms of the number of inputs and outputs it supports—as the LUT can be taken as storing only the truth table for a logic expression and not the expression itself, this is one of the more significant metrics. A LUT4 is a LUT with four inputs or outputs, and so could represent any combination of inputs and outputs provided that the total number does not exceed 4 and that there is at least a single output. For example, a LUT4 could represent a 3-input AND with 1 output, a 2-input logic expression with two outputs, or a 1-input expression with 3 inputs. Many LUTs are part of larger logic cells, which include common circuitry (such as carry logic or flip-flops) to aid in reducing the number of LUTs required for common functionality.

Instruments SN74LV4053A (2005b) triple 2-channel analogue (de)multiplexer can be had at £0.29 in single units, while the Nexperia HEF4067B (2016b) single-pole 16-throw switch is available at £1.44 in single quantities. In terms of practicality, the triple SPDTs (or 2:1 multiplexers) are likely to be the most useful, as they are a reasonable trade-off between limiting the ability of the microcontroller to scan its ADC channels and increasing circuit and firmware complexity.

• PAGE INTENTIONALLY LEFT BLANK •

Conclusion and Review

17. Critical evaluation

A general aim and set of objectives are given in chapter 1, with a set of specific requirements for each aspect of the project given in chapter 2. This chapter will evaluate whether, and to what extent, the aim, objectives, and requirements were met in carrying out the project.

17.1 Aim and objectives

The aim of the project, as set out in chapter 1.2, was to produce a prototype fan controller which would be readily convertible, with minimal further work, to a design suitable for commercial and volume production. This aim has been met.

The foundational elements of a fan controller are present in the proof-of-concept prototype (chapter 14), with the ancillary prototypes (chapter 15) being a concrete demonstration of what is required in relation to the more advanced aspects of a fan controller. The review of these prototypes and the application of that review in making recommendations for a production design (chapter 16) provides a base of work for a design suitable for commercial and volume production. Further, the information accumulated in the *Research and Theory* partition provides both a summarised reference and a starting point for further research. As such, both the components needed to fulfil the aim—that is, a prototype and an effort to reduce future workload—are present.

Additionally, chapter 1.2 lists eight general objectives. Each of these objectives is considered individually below. In summary, three objectives were entirely met, three were partially met, and two could not be definitively evaluated.

The technical knowledge objective

Objective 1.2.1 was to develop a product requiring a minimum amount of in-depth technical knowledge to use, and which would be usable with as little instruction as possible. This objective can be considered met.

In evaluating whether this objective is met, it is important to consider the person who would purchase or be interested in purchasing a fan controller—this person is likely to have some interest in information technology or an aspect of computer engineering—an uninterested person would be more likely to rely on the simple fan control provided by a motherboard—and is likely to be familiar with common components of a computer system. A production fan controller design would be aligned with the standards for these common components, and so an interested person would require little familiarisation. In particular, an interested person is likely to already be aware of 2- to 4-pin fan connectors and PCI Express 2 × 3 auxiliary power connectors, and is likely to be familiar with the 5.25" drive bay in which a production design would be mounted.

Further, an interested person is likely to be comfortable with the idea of a device not functioning before a device driver is installed, and is highly unlikely to be confused by the concept of the fan speed being controlled based on temperature. Although instruction would be required for more advanced concepts (such as the use of an extension to the drivers to enable the fan controller to control fans using another quantity), this feature is not required for the basic operation of the fan controller and so cannot affect whether the objective has been met.

An interested person may require simple instruction where an electrical concept is directly exposed—for example, where the drivers expose the ability to control the voltage supplied to a fan, the person may need to be informed of the relation between fan speed and supply voltage, and may require an explanation of the fan starting voltage—but such instruction could be summarised into single-sentence hints displayed in the user interface of the driver software.

The per-unit cost objective

Objective 1.2.2 was to attain as low a per-unit cost as practical without exceeding the £40 per-unit limit set out in requirement 2.1.8 and without excessive impact on the quality, safety, or reliability of the device. As the full production design for a fan controller was not produced, it cannot be said whether this objective has been met. It is, however, possible to make an estimate.

As requirement 2.1.8 specifies that the £40 per-unit cost is in reasonable volume, this estimate is made using the costs of components in 1000-unit quantities.

Following the recommendations in chapter 16.1, each fan connection would have a Fairchild FDMS7682 (£0.120) and a Nexperia NX7002AK (£0.022) as its control transistors. Voltage and current sensing remaining the same, and so there would be one TI INA180A2 (£0.180), one Vishay WSLP0805R0100FEA (£0.309) for a current-sense resistor, with one Vishay CRCW080538K3FKEB (£0.006) and one Vishay CRCW12101K10FKEAHP (£0.096) to form a potential divider. For the detection of fan functionality, a further NX7002AK (£0.022) enables sensing the presence of a pulled-up PWM control wire. The circuitry for conditioning the fan tachometer would use an ON NL37WZ17 (£0.113) Schmitt trigger with another potential divider formed by 68.1 kOhm and 348 kOhm resistors such as the Yageo RC0402FR-0768K1L (£0.002) and RC0402FR-07348KL (£0.002). Further, as the fan connection is standard, the Molex 47053-1000 (£0.161) would continue to be used. This gives an estimated cost per fan connection of £1.033, or £4.132 for four fan connections (representing around 10.3% of the maximum cost).

If the \$0.325/in^2 figure given in chapter 16.2 for the cost of manufacturing a circuit board is accurate, and assuming a circuit board size of 5.75”×4", the circuit board is likely to cost in the region of \$7.475—equivalent to £5.274 at the time of writing. This is approximately equivalent to 13.2% of the permissible per-unit cost, and gives a running total of around £9.406 (23.5%) when taken with the above cost.

As recommended in chapter 16.3, a Molex 45558-0003 (£0.301) would be used to connect to the 12 V supply and a typical four-position header, such as the Wurth Electronics 61300411121 (£0.068), to connect the 5 V supply. The 12 V connection would be controlled by a relay with a suitably low coil current, such as the Panasonic ADW1103HLW (£2.140), and the relay with a small transistor such as the Nexperia NX7002AK (£0.022). The 12 V supply would be protected by a typical non-resettable fuse, such as the Bel Fuse 0685H9200-01 (£0.134), and each fan by a resettable temperature-dependent fuse, such as the Bel Fuse 0ZCG0150BF2C (4 × £0.059), and a flyback diode such as the ComChip CDBA540-HF (4 × £0.093). Additionally, the 5 V supply would be protected by a Nexperia IP4220CZ6 (£0.093) transient voltage suppression array, by a resettable fuse such as a Littelfuse 0603L050SL (£0.579), and by a pair of Maxim MAX40200 (2 × £0.221) reverse polarity protection diodes. The cost for these components is then £4.387 (11%), giving a total of £13.793 (34.5%).

While this figure is not the full components cost—a production design would also be required to include, for example, decoupling capacitors for integrated circuits and resistors for the MOSFET gates—the cost of the further components needed is unlikely to be great, and so there may be as much as £25 remaining per unit for the metal sled (see chapter 16.2), fabrication, packaging, documentation, and the amortisation across multiple sales of costs such as device certification.[37] Further, as no in-depth component comparisons were done in making this estimate, it may be possible to find equally suitable components at a lower cost.

This considered, it is likely that this objective could, and would, be met.

The schedule objective

Objective 1.2.3 was to deliver the prototype in line with the proposed schedule set out in Appendix A1. As noted in Appendix A2, the proposed schedule did not accurately reflect the time available for the project, and so was not used after the 9th of November 2018. If the proposed schedule had been used, the three weeks allocated in chapter 3.1 for slippage would have been entirely used, and so the

[37] In order to be sold, a device must undergo formal certification. For an electronic device, this would generally involve testing by an accredited laboratory to ensure compliance with relevant European requirements and with the United States’ Federal Communications Commission requirements for electromagnetic compatibility. Additionally, the Windows operating system requires that driver packages be digitally signed (Microsoft Corporation, 2017a), and so an annual payment to a certificate authority would be necessary.

project may have run past the proposed deadline.

The proof-of-concept prototype was ordered on the 13th of November, and the proposed schedule estimated that the manufacturing and shipping would take no more than three weeks, which would mean that the prototype would have been received by the 4th of December. In reality, the prototype was received on the 21st of December—two weeks later than estimated. While other work was done in the time between the expected and actual date of receipt, it is unlikely that this other work would significantly lessen the slippage time used.

Further, the proposed schedule allocated three weeks for testing the prototype, the firmware for the prototype, and any software. It was not considered that there would be a significant limit on access to test equipment, and so the limit resulting in testing being carried out across the four weeks from the 12th of January to the 8th of February would have further eaten into the time available. No account is taken here of the time taken to test the ancillary prototypes, as these prototypes were not part of the proposed schedule.

As such, it cannot be said whether this objective has been met.

The development cost objective

Objective 1.2.4 was to develop the prototype referred to in the aim at a cost not greater than £600. This aim has been met, as the record of project expenditure in Appendix E shows that the total spend was £215.04—well below the limit.

The documentation objective

Objective 1.2.5 was to produce, to a high standard, all the materials which would be required to reproduce the prototype, and to include those materials in this report. This objective has been met.

Appendix F1 contains the bill of materials for the proof-of-concept prototype, as well as a list of the reference designators which identify these components in the schematic diagrams in Appendix F2 and the circuit design in Appendix F3. The discussion in Appendix F5 provides rationale for decisions made in producing the proof-of-concept firmware, while chapters 10 and 14 give reasoning for selecting the hardware used in the proof-of-concept prototype. Additionally, chapter 15 and Appendix G provide similar (but less detailed) discussion about the ancillary prototypes. These resources are sufficient to reproduce the project.

The test suite objective

Objective 1.2.6 was to produce for the prototype a robust suite of tests which was, where possible, automated. This objective has been partially met.

In relation to the proof-of-concept prototype, the objective has been met. A test plan is contained in Appendix F6, with a number of action items covering aspects of the proof-of-concept prototype hardware and firmware. The prototype had no software element. As automation was not considered practical, and given the concerns over the safety of the connection of the proof-of-concept prototype to the development kit, all tests were performed manually.

In relation to the ancillary prototypes, only the USB prototype was implemented practically, and so no tests could have produced for the sensors prototype. When considering the time available, it was considered that there was insufficient time to produce a suite of tests for the USB prototype by reason of the complexity of such a suite (which would be required to emulate the behaviour of the WinUSB interface and the USB device). As such, the USB prototype was tested manually and informally simultaneously with development, and so the objective cannot be considered met for USB prototype.

The design for manufacturing objective

Objective 1.2.7 was to have consideration for the ease of manufacturing the fan controller, and also to identify and implement accepted design for manufacturing practices (DFM). This objective has not been fully met.

As no specific design for manufacturing practices were identified, and so because there was no express consideration of those practices, this objective cannot be considered fully met. However, as some design for manufacturing practices were implemented, the objective can be partially met.

The most basic implementation of DFM was the designing of a circuit board (see Appendix F3) against the fabricator's listed capabilities. This process was largely automated, with the Autodesk EAGLE software used accepting design rule data provided by the fabricator and providing a warning if the design exceeded a limit or minimum set by the design rules. Examples of the fabricator's design rules are that a circuit trace cannot have a width less than 5 mil (0.127 mm), no feature on the circuit board can be closer than 10 mil (0.254 mm) from the edge of the circuit board, and no hole requiring a drill smaller than 12 mil (0.305 mm) can be drilled. Following these rules, the likelihood of manufacturing error is reduced.

Further, the proof-of-concept prototype used surface-mount components where possible, with only four through-hole components (one 20-pin header, one screw terminal, and two fan connectors). Similarly, the parts selected in chapter 16 are primarily surface-mount. This practice is recommended to reduce the additional work required by the assembler and to increase the speed of assembly, thereby reducing cost, with the use of through-hole parts for components experiencing considerable forces (such as external connectors, as is the case here) an accepted exception (Worthington Assembly, 2013a; 2013b).

Additionally, widely available parts were used or recommended for use where it was practical to do so. For example, 4000- or 7400-series logic devices were recommended for use with fan speed measurement; and circuits were designed around values from the standard series of resistors (BSI, 2015) where possible. Where a single-source component could not be avoided (as with the fan connectors or PCI Express auxiliary power connectors), the components were components either implementing a common standard or expressly referred to in that standard. This reduces the likelihood of a part becoming end-of-life or scarce having an impact on manufacturing (as the part could be replaced by a compatible alternative from another manufacturer).

Other DFM practices were also implemented.

The standards compliance objective

Objective 1.2.8 was to identify and comply with all standards relevant to the fan controller and all law applicable to it, and in doing so to have particular regard to standards and law concerned with health and safety and with requirements for electromagnetic compatibility. This objective has been largely met.

The *Research and Theory* partition identifies the majority of the standards which are relevant, with a list of those identified given in chapter 4.4. That chapter also identifies a number of relevant laws and discusses their applicability to prototype designs. Additional standards were identified in chapters 16.4 and 16.7.

The compliance of the prototypes with these standards is detailed throughout the report. For the standards and law listed in chapter 4.4, compliance with the 4-pin fan specification is detailed in chapters 5, 8.2, and 14.1; for SFF-8551J, discussion in chapters 7 and 16.2 covers the requirements for compliance; for the Universal Serial Bus specifications version 2.0 and 3.1, chapters 9.4, 11, 13, 16.4 contain the relevant discussion along with Appendix G1; and for the compliance with the PCI Express electromechanical specification for high-power cards, consideration in chapter 8.1 discusses requirements.

The compliance of the prototypes with the statutory instruments identified as relevant in chapter 4.4 is not discussed at length—while that chapter provides some discussion, the regulations were not believed to apply to the prototype, and so it was considered that attaining a functional prototype was of greater concern than compliance which was unnecessary and which would be prohibitively costly to verify. As such, while the requirements for identification and compliance with standards were largely fulfilled, the objective cannot be considered fully met.

17.2 Hardware requirements

In addition to the project aim and objectives, which were relatively general, a set of eight specific requirements were given in chapter 2.1 for hardware produced in carrying out the project. All of these requirements have been fulfilled.

As discussed in chapter 5.1, the two-, three-, and four-pin varieties of fan are the varieties commonly used in computer systems. The ability to control these types of fan is implemented in the proof-of-concept prototype (see chapter 8.2) and in the recommendations for a production design in chapter 16.1, and so everything needed to fulfil requirement 2.1.1 is present.

Regarding requirement 2.1.2, chapter 8.2 discusses the margins in the supply and concludes that there is sufficient margin to control four fans, while discussion of the resources required is included in chapters 14.2 and 16.1. While the proof-of-concept prototype only supported the control of two fans, suitable provision is made in the design and in recommendations for a production design to consider this requirement fulfilled.

To fulfil requirement 2.1.3, chapter 7 identifies and selects a suitable means for mounting a fan controller in a computer chassis, and chapter 16.2 contains further discussion on this subject. As before, it is considered that these recommendations are sufficient to fulfil the requirement without a practical implementation.

As discussed in chapters 8.1 and 16.3, the fan controller would use two supplies available internally to the host computer (and hence assumed to use the primary power supply of that computer). This fulfils requirement 2.1.4.

Requirement 2.1.5 was for the controller to interface with a host computer using standard and widely-available means. The selection in chapter 9 of USB, its use in the USB prototype (see chapter 15.1 and Appendix G1) and its recommendation in chapter 16.4 are sufficient to fulfil requirement 2.1.5.

Necessary to fulfil requirement 2.1.6 is a means of acquiring relevant data, which is provided by a temperature sensor (see chapter 14.1), transducers for the supply voltage and current, and means to receive the fan tachometer signal (both being discussed in chapter 16.1). As such, this requirement is fulfilled.

Requirement 2.1.7 was for the controller to comply with all applicable health and safety law. It is believed, based on resources provided by the Health and Safety Executive (2018), that chapter 4.4 covers the vast majority of applicable law, and that the requirements of this law have been met. In particular, it is believed that the general safety requirement would be fulfilled by the implementation of the protective measures discussed in chapter 16.3; that the requirements set out in the Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment Regulations 2012 (S.I. 2012/3032) ("RoHS") are fulfilled by the proof-of-concept prototype's use of only components advertised as RoHS-compliant by their manufacturers; and that, while it was not possible to verify the fan controller's compliance with electromagnetic compatibility requirements due to the prohibitive cost of doing so, it is unlikely that there would be significant issues in a production design. As such, while the requirement is not fulfilled in its entirety, it can be considered fulfilled.

In order to fulfil requirement 2.1.8, the cost of producing the production design for the fan controller could not exceed £40. As the section in chapter 17.1 on the per-unit cost objective discusses, while this cannot be evaluated without a full selection of components, it is likely that this requirement could be met, and so it can be considered fulfilled.

17.3 Firmware requirements

As well as the requirements for hardware discussed in chapter 17.2 above, a set of six firmware requirements was listed in chapter 2.2. All of these requirements have been fulfilled.

As stated in requirement 2.2.1, the firmware must be capable of controlling and monitoring each fan connected to the controller, and must be able to report fan speed to the host controller. These capabilities were implemented by the proof-of-prototype firmware (see chapter 14.4) and by the USB prototype (chapter 15.1 and Appendix G1), and so this requirement is fulfilled.

The proof-of-concept firmware also, in combination with the sensors prototype discussed in chapter 15.2 and Appendix G2, implements the interfacing with a set of transducers necessary to fulfil requirement 2.2.2.

The means of detecting the failure of a fan, as required to fulfil requirement 2.2.3, is discussed in chapter 16.1 in the section on monitoring fan functionality. The fan controller would be able to report such a failure as a disconnection event (see the discussion on protocol architecture in chapter 16.6), fulfilling the portion of the requirement on reporting failure and hence fulfilling requirement 2.2.3.

Requirement 2.2.4 is partially fulfilled by practical implementation and partially fulfilled by recommendation—the USB prototype (chapter 15.1 and Appendix G1) demonstrates what is necessary to communicate with the host computer, and the identification (in chapter 9.4) and recommendation (in chapter 16.7) of the use of the Device Firmware Upgrade (DFU) device class enables the in-circuit updating of the firmware in a production design.

In order to fulfil requirement 2.2.5, chapter 16.7 considers options for the storage of configuration data by the production design for a fan controller.

The framework necessary to support complex configurations (such as a function for the translating of temperature to fan speed) is demonstrated in chapter 15.1 and Appendix G1, which cover the USB prototype. Further, recommendations for changes to the host–controller protocol in chapter 16.6 consider other methods of implementing such a feature, fulfilling requirement 2.2.6.

17.4 Software requirements

Additional to the requirements for hardware and for firmware are three software requirements set out in chapter 2.3. All software requirements were fulfilled.

Requirement 2.3.1 was for the software to be capable of displaying data reported by the fan controller in real time. The USB prototype (refer to chapter 15.1 and Appendix G1) demonstrates the ability of software to display reported data, with the real-time aspect being enabled by the architectural changes to the protocol discussed in chapter 16.6. As such, this requirement is fulfilled.

The USB prototype also demonstrates the provision by the software to the fan controller a complex configuration prescribed by requirement 2.3.2. The ability of the software to provide a complex function—a mapping of a measured variable to a controlled variable, here temperature to fan speed—fulfils this requirement, as this ability includes providing simple and complex configuration data.

To fulfil requirement 2.3.3, it was necessary for the software to support updating the fan controller firmware. Although no practical demonstration of this feature was given, chapter 16.7 covers what would be needed to support this feature, and so this requirement can be considered fulfilled.

18. References

1. ADDA Corporation, 2004. *Specification for Approval – Model No. AD0912-A70GL (TC) DC Fan (Lead Free).* rev. A ed. s.l.:ADDA Corporation.

2. ARM, 2010. *Cortex-M4 Devices – Generic User Guide.* Cambridge: ARM Limited.

3. ASTM, 2002. *ASTM B258-02: Standard Specification for Standard Nominal Diameters and Cross-Sectional Areas of AWG Sizes of Solid Round Wires Used as Electrical Conductors.* West Conshohocken(PA): ASTM International.

4. Axelson, J., 2009. *USB Complete: The Developer's Guide.* 4th ed. Madison(Wisconsin): Lakeview Research LLC.

5. Bradner, S., 1997. *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels.* [Online]
   Available at: https://tools.ietf.org/html/rfc2119

6. BSI, 1995. *BS EN ISO/IEC 7498-1: Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model.* s.l.:British Standards Institution.

7. BSI, 1999. *BS ISO/IEC 9899: Programming languages — C.* s.l.:British Standards Institution.

8. BSI, 2015. *BS EN 60063: Preferred number series for resistors and capacitors.* s.l.:British Standards Institution.

9. Central, 2013. *CFSH05-20L: Surface Mount Silicon Low Vf Schottky Diode – 0.5 amp, 20 volt.* R2 ed. s.l.:Central Semiconductor Corp..

10. Cooler Master Co., Ltd, 2008. *MegaFlow 200 Blue LED Silent Fan.* s.l.:Cooler Master Co., Ltd.

11. Dickens, C., 2017. *libusb.* [Online]
    Available at: http://libusb.info/
    [Accessed 17 March 2018].

12. EIA, 1969. *EIA RS-232-C: Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interface.* Washington D.C.: Electronic Industries Association.

13. Fairchild, 2015a. *FDMS7682: N-Channel PowerTrench MOSFET – 30 V, 6.3 mOhm.* s.l.:ON Semiconductor.

14. Fairchild, 2015b. *S1A–S1M: General-Purpose Rectifiers.* rev. 2.10 ed. s.l.:ON Semiconductor.

15. Fluke, 2007. *Dual impedance digital multimeters – What's the point?.* rev. B ed. Eindhoven: Fluke Corporation.

16. FTDI Ltd., 2010. *Can I use FTDI's VID for my own product?.* [Online]
    Available at:
    http://www.ftdichip.com/Support/Knowledgebase/index.html?caniusef tdisvidformypr.htm
    [Accessed 15 September 2017].

17. Fujitsu, 2015. *MB85RS16N: 16 K (2 K × 8) Bit SPI.* Yokohama(Kanagawa): Fujitsu Semiconductor.

18. Gamazo-Real, J. C., Vázquez-Sánchez, E. & Gómez-Gil, J., 2010. Position and Speed Control of Brushless DC Motors Using Sensorless Techniques and Application Trends. *Sensors,* 19 July.10(7).

19. Giesselmann, M., Salehfar, H., Toliyat, H. A. & Rahman, T. U., 2002. Modulation Strategies. In: S. L. Timothy, ed. *The Power Electronics Handbook.* Boca Raton(Florida): CRC Press, p. 305–339.

20. Health and Safety Executive, 2018. *UK law on the design and supply of products.* [Online]
Available at: http://www.hse.gov.uk/work-equipment-machinery/uk-law-design-supply-products.htm
[Accessed 31 March 2018].

21. HM Government, 2005. *The General Product Safety Regulations (S.I. 2005/1803).* s.l.:National Archives.

22. HM Government, 2010. *The Ecodesign for Energy-Related Products Regulations 2010 (S.I. 2010/2617).* s.l.:National Archives.

23. HM Government, 2012. *The Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment Regulations 2012 (S.I. 2012/3032).* s.l.:National Archives.

24. HM Government, 2016a. *The Electrical Equipment (Safety) Regulations 2016 (S.I. 2016/1101).* s.l.:National Archives.

25. HM Government, 2016b. *The Electromagnetic Compatibility Regulations 2016 (S.I. 2016/1091).* s.l.:National Archives.

26. Honeywell, n.d. *Hall Effect Sensing and Application.* s.l.:Honeywell.

27. Horowitz, P. & Hill, W., 1989. *The Art of Electronics.* 2nd ed. Cambridge: Cambridge University Press.

28. IEEE, 2015. *IEEE Std. 802.3: Standard for Ethernet.* 2015 ed. New York: IEEE Standards Association.

29. Intel Corporation, 2002. *ATX Specification.* v2.1 ed. s.l.:Intel Corporation.

30. Intel Corporation, 2005a. *4-Wire Pulse Width Modulation (PWM) Controlled Fans Specification.* rev. 1.3 ed. s.l.:Intel Corporation.

31. Intel Corporation, 2005b. *ATX12V Power Supply Design Guide.* v2.2 ed. s.l.:Intel Corporation.

32. Intel Corporation, 2005c. *Front Panel I/O Connectivity Design Guide.* rev. 1.3 ed. s.l.:Intel Corporation.

33. Intel Corporation, 2007. *Intel Core 2 Duo processor with the Mobile Intel 945GME Express Chipset Development Kit User's Manual.* rev. 001 ed. s.l.:Intel Corporation.

34. Intel Corporation, 2013. *Desktop boards — Three-wire and four-wire fan connectors.* [Online]
Available at:
https://web.archive.org/web/20140122004200/http://www.intel.com/support/motherboards/desktop/sb/cs-012074.htm
[Accessed 8 September 2017].

35. IPC, 1999. *IPC-SM-782A (including Amendments 1 and 2): Surface Mount Design and Land Pattern Standard.* Northbrook(Illinois): IPC.

36. ITU, 2012. *ITU-T X.667 (ISO/IEC 9834-8): Information technology – Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers.* 3.0 ed. s.l.:International Telecommunication Union, Telecommunication Standardization Sector.

37. Karki, J., 1998. *Understanding Operational Amplifier Specifications.* Austin(Texas): Texas Instruments.

38. Keagy, M., 2002. *Calculate Dissipation for MOSFETs in High-Power Supplies.* [Online]
Available at: http://electronicdesign.com/boards/calculate-dissipation-mosfets-high-power-supplies
[Accessed 12 September 2017].

39. Krakauer, D., 2011. *Anatomy of a Digital Isolator.* Norwood(Massachusetts): Analog Devices.

40. Lattice, 2016. *iCE40 UltraLite™ Family Data Sheet.* version 1.4 ed. s.l.:Lattice Semiconductor.

41. Leach, P. J., Mealling, M. & Salz, R., 2005. *RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace.* s.l.:Internet Engineering Task Force.

42. Lin, Z. & Pearson, S., 2013. *An inside look at industrial Ethernet communication protocols,* Dallas: Texas Instruments.

43. Maxim, 2011. *MAX11638/MAX11639/MAX11642/MAX11643: 8-Bit, 16-/8-Channel, 300ksps ADCs with FIFO and Internal Reference.* rev. 0 ed. s.l.:Maxim Integrated.

44. Maxim, 2017. *MAX40200: Ultra-Tiny Micropower, 1A Ideal Diode with Ultra-Low Voltage Drop.* rev. 1 ed. s.l.:Maxim Integrated.

45. Microchip Technology, 2017. *Application to Request a Sublicense of Microchip's Universal Serial Bus Vendor ID.* [Online]
Available at: http://www.microchip.com/usblicensing/
[Accessed 15 September 2017].

46. Microsoft Corporation, 2007. *Microsoft OS Descriptors Overview.* 1.0 ed. s.l.:Microsoft.

47. Microsoft Corporation, 2009. *How does USB stack enumerate a device?.* [Online]
Available at:
https://blogs.msdn.microsoft.com/usbcoreblog/2009/10/30/how-does-usb-stack-enumerate-a-device/
[Accessed 27 January 2018].

48. Microsoft Corporation, 2017a. *Get a code signing certificate.* [Online]
Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/dashboard/get-a-code-signing-certificate
[Accessed 28 March 2018].

49. Microsoft Corporation, 2017b. *Microsoft OS 2.0 Descriptors Specification.* 2.0 ed. s.l.:Microsoft.

50. Microsoft Corporation, 2017c. *Overview of Device Interface Classes.* [Online]
Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/overview-of-device-interface-classes
[Accessed 18 February 2018].

51. Microsoft Corporation, 2017d. *USB-Specific UMDF 1.x Interfaces.* [Online]
Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf/usb-specific-umdf-1-x-interfaces
[Accessed 17 March 2018].

52. Microsoft Corporation, 2017e. *WinUSB (Winusb.sys) Installation.* [Online]
Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/winusb-installation#a-href-idinfawriting-a-custom-inf-for-winusb-installation
[Accessed 8 March 2017].

53. Microsoft Corporation, n.d. *WinUSB Device.* [Online]
Available at: https://msdn.microsoft.com/en-gb/library/windows/hardware/hh450799.aspx
[Accessed 10 October 2017].

54. Morris, C., 2016. *Blu-Ray Struggles in the Streaming Age.* [Online]
Available at: http://fortune.com/2016/01/08/blu-ray-struggles-in-the-streaming-age/
[Accessed 13 September 2017].

55. NetApplications.com, 2018. *Operating System Share by Version — February 2017 to February 2018.* [Online]
Available at: https://netmarketshare.com/operating-system-market-share.aspx
[Accessed 17 March 2018].

56. Nexperia, 2011a. *IP4220CZ6: Dual USB 2.0 integrated ESD protection.* rev. 5 ed. s.l.:Nexperia.

57. Nexperia, 2011b. *TDZxJ series: Single Zener diodes.* rev. 2 ed. s.l.:Nexperia.

58. Nexperia, 2015. *NX7002AK: 60 V, single N-channel Trench MOSFET.* v.7 ed. s.l.:Nexperia.

59. Nexperia, 2016a. *HEF4053B: Triple single-pole double-throw analog switch.* rev. 12 ed. s.l.:Nexperia.

60. Nexperia, 2016b. *HEF4067B: 16-channel analog multiplexer/demultiplexer.* rev. 8 ed. s.l.:Nexperia.

61. Nexperia, 2016c. *HEF4104B: Quad low-to-high voltage translator with 3-state outputs.* rev. 9 ed. s.l.:Nexperia.

62. Nisarga, B., 2011. *PWM DAC Using MSP430 High-Resolution Timer.* s.l.:Texas Instruments.

63. Noctua, 2017a. *NF-A20 PWM Premium Fan.* s.l.:Rascom Computerdistribution Ges.m.b.H.

64. Noctua, 2017b. *Smooth Commutation Drive.* [Online] Available at: http://noctua.at/en/smooth-commutation-drive [Accessed 9 September 2017].

65. NXP, 2014. *UM10204: I^{2}C-bus specification and user manual.* rev. 6 ed. s.l.:NXP Semiconductors.

66. NXP, 2017a. *Application to use NXP Semiconductors USB-IF Vendor Identification Number.* [Online] Available at: https://contact.nxp.com/vid-use-app [Accessed 15 September 2017].

67. NXP, 2017b. *PCT2075: I^{2}C-bus Fm+, 1 °C accuracy, digital temperature sensor and thermal watchdog.* rev. 10 ed. s.l.:NXP Semiconductors.

68. ON Semiconductor, 2005. *TVS/Zener Theory and Design Considerations Handbook.* rev. 0 ed. Phoenix(Arizona): ON Semiconductor.

69. ON Semiconductor, 2013. *NL37WZ17: Triple Noninverting Schmitt-Trigger Buffer.* rev. 8 ed. s.l.:ON Semiconductor.

70. ON Semiconductor, 2014. *MC14504B: Hex Level Shifter for TTL to CMOS or CMOS to CMOS.* rev. 9 ed. Denver(Colorado): ON Semiconductor.

71. OpenMoko Inc., 2017. *USB Product IDs.* [Online] Available at: http://wiki.openmoko.org/wiki/USB_Product_IDs [Accessed 15 September 2017].

72. Otander, J., 2015. *Welcome to pid.codes.* [Online] Available at: http://pid.codes/pidcodes/2015/04/03/welcome/ [Accessed 2015 September 2017].

73. PCI-SIG, 2008. *PCI Express® 225 W/300 W High Power Card Electromechanical Specification.* rev. 1.0 ed. s.l.:PCI-SIG.

74. PCI-SIG, 2010. *PCI Express® Base Specification – Revision 3.0.* s.l.:PCI-SIG.

75. PCI-SIG, 2017. *Become a Member.* [Online] Available at: http://pcisig.com/membership/become-member [Accessed 15 September 2017].

76. SATA-IO, 2009. *Serial ATA Revision 3.0.* s.l.:Serial ATA International Organization.

77. SD Association, 2017. *SD Specifications — Part 1: Physical Layer Simplified Specification.* v6.00 ed. s.l.:SD Card Association.

78. SD Association, 2018. *How to Start Using SD Standards in Your Product.* [Online]
Available at: https://www.sdcard.org/developers/howto/index.html
[Accessed 23 March 2018].

79. SFF, 2000. *SFF-8551J — Form Factor of 5.25" CD Drives.* rev. 3.3 ed. s.l.:Storage Network Industry Association — SFF Technology Affiliate Technical Working Group.

80. Silicon Labs, 2013a. *AN0019: EEPROM Emulation.* rev. 1.09 ed. Austin(Texas): Silicon Labs.

81. Silicon Labs, 2013b. *AN0024: EFM32 Pulse Counter.* rev. 1.07 ed. Austin(Texas): Silicon Labs.

82. Silicon Labs, 2013c. *AN0046: USB Hardware Design Guide.* rev. 1.01 ed. Austin(Texas): Silicon Labs.

83. Silicon Labs, 2013d. *User Manual: Starter Kit EFM32WG-STK3800.* Austin(Texas): Silicon Labs.

84. Silicon Labs, 2014a. *EFM32WG Reference Manual.* rev. 1.0 ed. Austin(Texas): Silicon Labs.

85. Silicon Labs, 2014b. *EFM32WG990 Datasheet — F256/F128/F64.* rev. 1.40 ed. Austin(Texas): Silicon Labs.

86. Silicon Labs, 2017a. *AN0014: EFM32 Timers.* rev. 1.10 ed. Austin(Texas): Silicon Labs.

87. Silicon Labs, 2017b. *EFM32 Wonder Gecko Software Documentation: CMU.* [Online]
Available at:
https://siliconlabs.github.io/Gecko_SDK_Doc/efm32wg/html/group__CMU.html
[Accessed 3 February 2018].

88. Silicon Labs, 2017c. *Request a Product ID (PID).* [Online]
Available at: https://www.silabs.com/products/interface/request-product-id
[Accessed 15 September 2017].

89. ST, 2017. *M24C16-W, M24C16-R, M24C16-F: 16-Kbit I^2C bus EEPROM.* rev. 9 ed. s.l.:ST Microelectronics.

90. StarTech.com Ltd, 2018a. *18in Internal 5-pin USB IDC Motherboard Header Cable – F/F.* [Online]
Available at: https://www.startech.com/uk/Cables/USB-2.0/Internal-and-Panel-Mount/18in-Internal-5-pin-USB-IDC-Motherboard-Header-Cable~USBINT5PIN
[Accessed 10 March 2018].

91. StarTech.com Ltd, 2018b. *6in USB 2.0 Cable – USB A Female to USB Motherboard 4-pin Header F/F.* [Online]
Available at: https://www.startech.com/uk/Cables/USB-2.0/Internal-and-Panel-Mount/6-USB-A-Female-to-Motherboard-Header-Adapter~USBMBADAPT
[Accessed 10 March 2018].

92. StatCounter, 2018. *Desktop Windows Version Market Share Worldwide — February 2018.* [Online]
Available at: http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide
[Accessed 17 March 2018].

93. Sweney, M., 2017. *Film and TV streaming and downloads overtake DVD sales for first time.* [Online]
Available at: https://www.theguardian.com/media/2017/jan/05/film-and-tv-streaming-and-downloads-overtake-dvd-sales-for-first-time-netflix-amazon-uk
[Accessed 13 September 2017].

94. Tan, L. & Jiang, J., 2013. *Digital Signal Processing: Fundamentals and Applications.* 2nd ed. Oxford: Academic Press.

95. Texas Instruments, 2003. *CD40109B Types: CMOS Quad Low-to-High Voltage Level Shifter.* rev. B ed. Dallas(Texas): Texas Instruments.

96. Texas Instruments, 2005a. *CD54HC4049, CD74HC4049, CD54HC4050, CD74HC4050 High-Speed CMOS Logic Hex Buffers, Inverting and Non-Inverting.* rev. I ed. Dallas(Texas): Texas Instruments.

97. Texas Instruments, 2005b. *SN54LV4053, SN74LV4053A: Triple 2-Channel Analog Multiplexers/Demultiplexers.* s.l.:Texas Instruments.

98. Texas Instruments, 2016. *LC Filter Design.* A ed. Austin(Texas): Texas Instruments.

99. Texas Instruments, 2017. *INAx180 Low- and High-Side Voltage Output, Current-Sense Amplifiers.* rev. B ed. Austin(Texas): Texas Instruments.

100. Texas Instruments, n.d. *Application to Use Texas Instruments Incorporated Universal Serial Bus Vendor ID.* [Online]
Available at:
http://focus.ti.com/en/download/mcu/application_for_sublicense.pdf
[Accessed 15 September 2017].

101. USB-IF, 1997. *Universal Serial Bus Common Class Specification.* rev. 1.0 ed. s.l.:Universal Serial Bus Implementers Forum.

102. USB-IF, 2000. *Universal Serial Bus Specification.* rev. 2.0 ed. s.l.:Universal Serial Bus Implementers Forum.

103. USB-IF, 2004. *Universal Serial Bus Device Class Specification for Device Firmware Upgrade.* v1.1 ed. s.l.:Universal Serial Bus Implementers Forum.

104. USB-IF, 2010. *Universal Serial Bus Mass Storage Class — Specification Overview.* rev. 1.4 ed. s.l.:Universal Serial Bus Implementers Forum.

105. USB-IF, 2013. *Universal Serial Bus 3.1 Specification.* rev. 1.0 ed. s.l.:Universal Serial Bus Implementers Forum.

106. USB-IF, 2016. *USB Class Codes.* [Online]
Available at: http://www.usb.org/developers/defined_class
[Accessed 15 October 2017].

107. USB-IF, n.d. *Getting a Vendor ID.* [Online]
Available at: http://www.usb.org/developers/vendor/
[Accessed 15 September 2017].

108. Walters, K., 2010. *Zeners and Transient Voltage Suppressors: Can Either Device Be Used For The Same Applications?.* rev. 0 ed. Scottsdale(Arizona): Microsemi.

109. Worthington Assembly, 2013a. *Design for Manufacturability (DFM): Use Surface Mount Components (SMT) – Part 1 of Many.* [Online]
Available at:
https://www.worthingtonassembly.com/blog/2013/03/22/design-for-manufacturability-dfm-use-surface-mount-components-smt-part-1-of-many
[Accessed 30 March 2018].

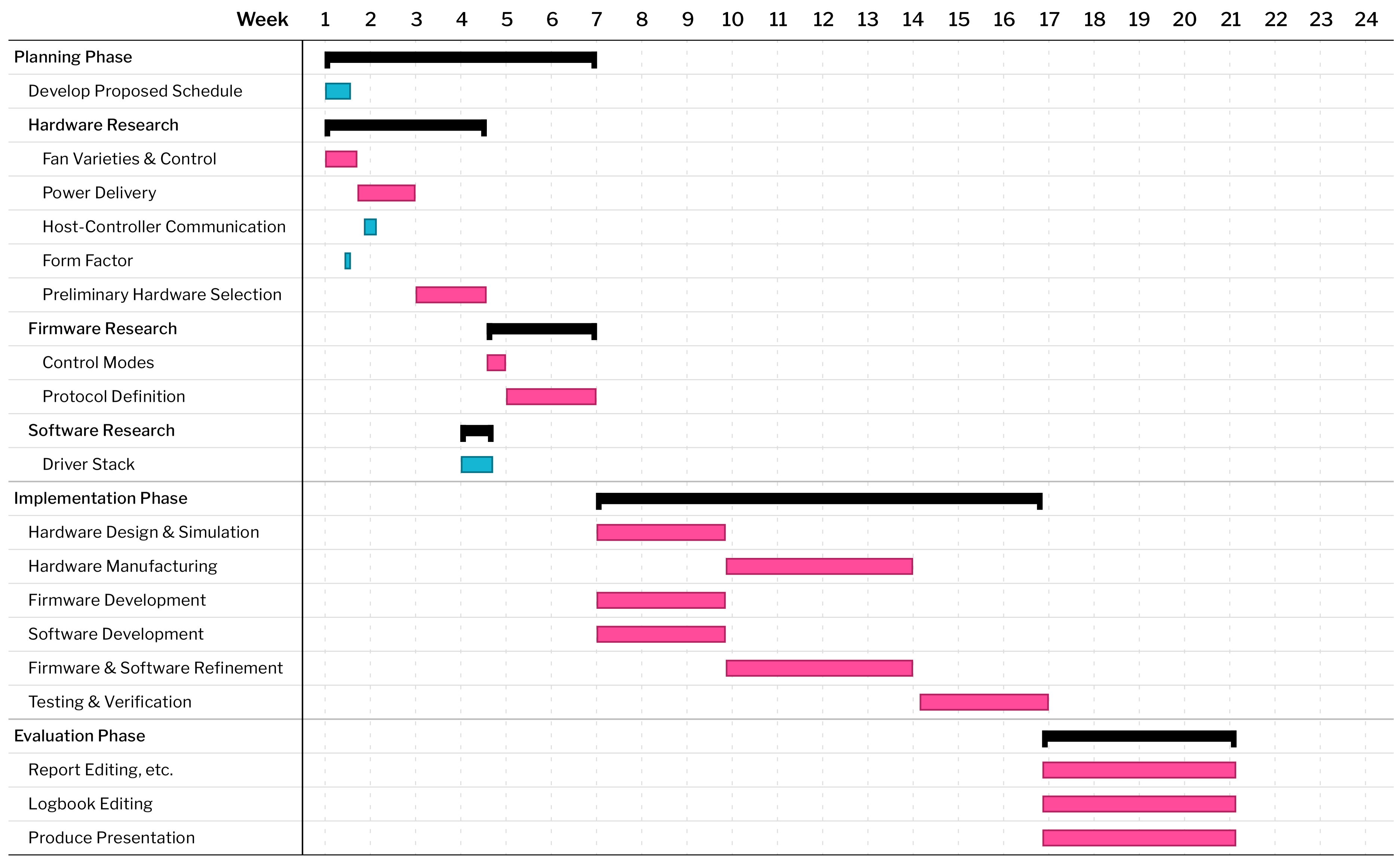
110. Worthington Assembly, 2013b. *Design for Manufacturability (DFM): Use Surface Mount Components (SMT) – Part 3 of Many.* [Online]
Available at:
https://www.worthingtonassembly.com/blog/2013/03/26/design-for-manufacturability-dfm-use-surface-mount-components-smt-part-3-of-many
[Accessed 30 March 2018].

19. Figures

20. Tables

Appendix A
Project schedule

A1 PROPOSED SCHEDULE



A2 Revisions to the proposed schedule

The proposed schedule was based around a 24-week period of work. Originally, it was believed that the implementation phase was to be completed on or before the 22th of December 2017, with a final deadline on the 23rd of February 2018. However, as noted in the progress log entry for the 9th of November 2017, these dates were instead a general recommendation. From that date, the project ceased to follow the proposed schedule.

The true target date for completion was then known to be the 20th of April 2018 and, although this was not recorded, it was planned to use this additional time to consider other relevant topics. Despite the proposed schedule no longer applying to the project, work continued largely following the list of tasks in that schedule until Thursday the 15th of February 2018.

On the 15th of February 2018, the remaining work and the remaining time were reviewed. It was considered that a reasonable target for completion of the project implementation was mid to late March 2018. This was the target worked to from that date.

In March 2018, it was clarified that the presentation portion of the project would not need to be completed at the same time as the report (as was assumed when the proposed schedule was produced).

This target was further reviewed on the 7th and on the 20th of March. No issues with the schedule were foreseen, and work was progressing as scheduled.

On the 23rd of March, the implementation phase was provisionally completed.

• PAGE INTENTIONALLY LEFT BLANK •

Appendix B
Progress log

2017

SEPTEMBER

Thursday, 7th
Initial requirements specification completed, comprising: a brief with context, aim, and objectives; requirements for each aspect of the project; and a proposed schedule for the completion of the project.

Friday, 8th
Progress log started.

Gantt chart for the proposed schedule (see Appendix A1) started.

Research started, with the findings of research into the available and common varieties of computer fan added to the "Research and Theory" partition.

Saturday, 9th
Gantt chart for the proposed schedule (see Appendix A1) completed.

Research into the construction and control of fans added.

Sunday, 10th
The section on research into the control of fans provisionally completed. An explanatory section on pulse width modulation and a section for research into fan monitoring started.

Monday, 11th
Further research into control and monitoring re PWM control being unusable for certain varieties of fan (as a result of the commonness of Hall effect sensors), and alternative speed control strategies.

Form factor research put on hold as it is relatively minor and not expected to take significant or substantial time to complete.

Tuesday, 12th
Research into fan power requirements and power delivery added. This is in line with the proposed schedule, where fan variety and control research would be completed by the 12th with power delivery research starting on the 13th.

Wednesday, 13th
Research into device form factors added, and form factor for the controller selected with justification.

Thursday, 14th — Week 1
Research into host–controller communications interfaces started.

Friday, 15th

Research into host–controller communications interfaces completed. USB is selected as the interface to be used.

This is largely in line with the proposed schedule, where host–computer interfaces research would be completed by the 15th. There has been slight slippage as little research into power delivery has been done. However, there is still nearly a week until power delivery research is to be completed.

Saturday, 16th

Research into the power supplies available in a typical computer.

Sunday, 17th

Sources of power available in a typical computer identified, and the most appropriate sources for the controller selected. The controller is to use a combination of PCI Express auxiliary power connectors and the power provided over the USB host–controller interface.

Monday, 18th

Confirming that the current available from the PCI Express auxiliary power connector is sufficient to power four typical computer fans.

In findings summary, started a list of applicable standards.

Tuesday, 19th

Research into controlling fan supply voltage started.

Thursday, 21st Week 2

Research revealed that use of a low-pass filter to "smooth" a pulse-width modulated waveform to approximately its average voltage could provide the means to control the speed of 2-/3-pin fans.

Potential challenges with this method were the voltage drop in the filter and in the transistor controlling the 12 V supply, causing the actual supply voltage to drop significantly below 12 V.

Friday, 22nd

Following on from the findings made yesterday, a bipolar junction transistor (BJT) in a common collector topology with the base driven by the output of a low-pass filter was identified as a simple method of control.

Work on this is likely on encroach into the time allocated in the proposed schedule for preliminary parts selection, however seven days was allocated to that task as a buffer. While this development is not in line with the proposed schedule, it was anticipated.

Saturday, 23rd

Majority of simulation/etc. work completed, but results must be converted into a form suitable for inclusion in the report.

Sunday, 24th

Results now in form suitable for inclusion in the report, and majority of discussion on power delivery completed. Some minor discussion remains.

There should be insignificant impact on the “preliminary hardware selection” task, which the proposed schedule lists as being completed by the 2nd October.

Monday, 25th

Minor discussion on power delivery completed. May require slight updates or additions as other aspects of the project are explored, but it is not anticipated that any further significant power delivery-related work will be required in the planning phase.

Tuesday, 26th

A minor chapter on pulse width modulation (PWM) started on Sunday, 10th completed.

Wednesday, 27th

Summary of findings updated. Outline for chapter on preliminary hardware section added.

Thursday, 28th Week 3

Rationale for the selected controller added. The selection of the precise model of controller is deferred until later, however the Silicon Labs EFM32WG family was selected, largely as a result of a development kit for the family already being to hand. It is acknowledged that this might not be the most economical selection, but the family does fulfil the criteria set out.

Friday, 29th

Begin introduction to power transistor portion of preliminary hardware selection.

Saturday, 30th

Complete the introduction to power transistor portion of preliminary hardware selection.

OCTOBER

Sunday, 1st

Further work on power transistor preliminary hardware selection. Main considerations about transistors completed, with heat-sinking considerations in progress. Once heat-sinking considerations are completed, a final portion on the preliminary selection bringing together the considerations can be done.

The “driver stack determination” task on the proposed schedule is unlikely to be completed by the second, but additional time was allocated to the “control modes determination” task for overrun. The “control modes” task, which is concerned with the various user-selectable modes of fan control, is not anticipated to take the three days scheduled for it.

Monday, 2nd

Begin introduction to driver stack determination chapter.

Tuesday, 3rd

Further research indicated that the previous selection of a common collector BJT to act as a voltage buffer in the controlling of the voltage supplied to the fan is not viable. The transistor would dissipate too much heat to be cooled by any reasonably-sized passive heatsink.

Identified alternative is the use of the PWM DAC with filter directly to provide power to the fan. Further required work on this is the researching of the information required for the biasing/etc. of any transistors, as well as preliminary hardware selection. However, it is anticipated that preliminary hardware selection for this revised method will be simpler.

This is a significant setback and is likely to affect the proposed schedule. Significant work must be done before the 6th in order for the critical path to remain on-schedule. However, as the proposed schedule did not include weekends in its allocated time, this may be feasible.

Wednesday, 4th

Discussion of voltage control technique provisionally completed, next step is to replace the preliminary hardware selection discussion.

Thursday, 5th Week 4

An RLC filter is provisionally identified as the type of filter for the PWM DAC. It is anticipated that there will be overrun past the 6th, but current progress is good and, barring any further setbacks, it is not anticipated that there will be great delays. There is a chance that delays here will result in further delay at the end of the “protocol definition” task, but the next task after that is “hardware design/simulation” which, while not simple, is unlikely to throw up any unexpected hurdles.

Friday, 6th

Component selection for the PWM DAC started. Basics identified, more specific selection to be done. Unlikely to be as involved as the selection for the previous method would have been, as most of the components are fairly generic.

Delays still anticipated.

Saturday, 7th

Majority of discussion on the selection of components for the PWM DAC done, with the remainder of work on the PWM DAC being the selection of a diode for use in a voltage snubber. There is some uncertainty about whether the PWM DAC will work as intended, but that work will have minimal impact on the critical path. Any testing can be done with a simple test assembly in parallel with other tasks, and—if the PWM DAC is found not to work—use of PWM directly would be an acceptable, if less preferable, alternative. Use of PWM directly would require simple hardware design adjustments—largely the removal of components only. If time permits, the effect of PWM on the fan tachometer output could also be tested to determine whether the output could be conditioned into a usable signal.

Once that is completed, research into the driver stack must be done. After that, work on defining the protocol for the fan controller must be done. It is expected that all of this can be completed by the time the next task (hardware design/simulation) is scheduled to start on the 20th October.

Monday, 9th

The Nexperia TDZ12J is selected as the Zener diode for the snubber circuit. The discussion related to the justification for this is also completed.

Tuesday, 10th

Discussion, selection of the driver stack completed.

Wednesday, 11th

Begin discussion on control modes. This is not expected to take considerable time, and should be largely (if not entirely) complete before the 13th, leaving slightly longer than a week for the protocol definition.

Thursday, 12th Week 5

Control modes discussion provisionally complete. Relatively minor changes and additions made to discussion related to the PWM DAC and snubber circuit.

Friday, 13th

Reading up on USB to refresh knowledge.

Saturday, 14th

Begin USB-related discussion. Basic plan for the protocol laid out, but specifics not yet committed to paper.

Sunday, 15th

Complete discussion about how USB devices are identified generally, and in the specific case of the fan controller. Begin drafting protocol specification.

Monday, 16th

Add provisionally-complete discussion relating to communication with USB devices and the correct definition of non-standard requests and descriptors for a USB device. Minor other work done on protocol specification.

Tuesday, 17th

Framework of the device protocol laid out, finer details to be established.

Wednesday, 18th

Minor work, basics of two USB requests specified.

Thursday, 19th — Week 6

USB protocol specification provisionally completed. It is expected that there will be a need to make modifications as the implementation of the protocol reveals issues which could not have otherwise been identified.

Friday, 20th

Familiarisation with Autodesk EAGLE schematic capture and layout software.

Process of drafting a preliminary bill of materials for the components identified during preliminary hardware selection revealed that sourcing or using a 7.8 mH inductor is infeasible. Adequately-rated inductors around 7.8 mH are comparable in size to a fist, weigh multiple kilograms, and tend to cost in excess of £20. However, reducing inductance to a feasible level and increasing capacitance to compensate was determined in simulation to produce a comparable result.

Minor rework and recalculations expected to carry into tomorrow, but no significant impact on other work anticipated.

Saturday, 21st

Reworking complete.

Begin the design of the proof-of-concept circuit. The final circuit cannot be made without first confirming that the PWM DAC works as-designed. A prototype board is to be designed and manufactured and used with the microcontroller development kit to verify correct operation.

Sunday, 22nd

Begin write-up on proof-of-concept circuit and component selection for the ancillary circuit components that will be required for the proof-of-concept (but which were not covered in the preliminary selection).

Monday, 23rd

Requirements for flyback diode set out.

Tuesday, 24th

Flyback diode selected as the Comchip CDBA540-HF.

Wednesday, 25th

Update proof-of-concept prototype schematic diagram with selected diode, and begin development kit connection-related work.

The pins for the microcontroller selected for use and the microcontroller on the development kit are multifunction. A limited number of these pins are exposed via a pin header on the development kit, and so correct pin use selection is vital for ensuring that the desired functionality can be tested.

Thursday, 26th — Week 7

Further work on component selection for the proof-of-concept circuit, and work on writing up those selections. Tweaks made to previously written-up sections.

Friday, 27th

Selection and related discussion provisionally complete for the PWM DAC capacitor, power connector, and fan header for the proof-of-concept circuit. It is not expected that there will be much more selection work until the design of any final circuit.

Saturday, 28th

Selection and related discussion started, largely completed, for the interface to the fan tachometer output. Remaining work is largely to do with the comparison of potential devices.

Further development kit connection-related work done. Modifications made as required to schematic diagram and circuit layout symbols in EAGLE, and the start of an exhaustive list of required pin functions is included in the report. The list is to be completed, and follow-up work from that is the selection of appropriate connections to the development kit and the updating of the circuit schematic accordingly.

Sunday, 29th

The fan tachometer interface selection work is completed. The device selected is the Texas Instruments CD74HC4050M96 (or equivalent), to be used to shift the voltage level of the tachometer from 12 V to the 3.3 V desired.

Monday, 30th

Corrections made to calculations relating to the selection of the PWM DAC control transistor. A voltage drop was substituted where a resistance should have been, resulting in a significantly higher calculated power.

Discussion and selection for the 4-pin fan PWM control interface provisionally completed. A fan requires the controller to provide an open-collector or open-drain output, and the FDMS7682 selected for use in the PWM DAC is a more than suitable choice (if not the most economical option).

Tuesday, 31st

Begin discussion of the monitoring transducers for the fans. Those transducers being the devices which are to allow the fan controller to monitor the current and voltage provided for each fan.

Preliminary research shows that monolithic current transducers are expensive, and so it is more likely that current will be measured by means of the voltage across a current-sense resistor being fed into an op-amp. Voltage monitoring is not anticipated as being a challenge.

NOVEMBER

Wednesday, 1st

Minor updates to the exhaustive list of required pin functions.

Thursday, 2nd Week 8

The Texas Instruments INA180A2IDBVT current-sense op-amp is selected as the current transducer for use in monitoring currents.

A ratio of 4:1 is selected for the potential divider to be used in measuring the fan voltage, with absolute resistances of 20 and 5 kiloohms. The potential divider is a simple means of reducing the voltage so that it can be safely fed to the analogue-to-digital converter (ADC) on the microcontroller.

The list of required pin functions is provisionally given as three PWM functions, four ADC channels or functions, and two pulse-counter functions.

Friday, 3rd

Preliminary pin selections for each required function made.

Begin creating parts diagrams for eventual use in circuit schematics. Updates made to the schematic for the proof-of-concept prototype.

Saturday, 4th

Existing schematic diagrams updated, and schematics for the board-to-external connections, PWM DAC-controlled fan connection, and bare fan connection produced. These schematics provisionally complete.

Begin on design rationales for small notes that were not appropriate in the main hardware selection section. Two such rationales completed.

Sunday, 5th

Add rationale note for the placement of the voltage transducer for the PWM DAC-controlled fan on the prototype, and correct the relevant portion of the schematic (which had shown this connection incorrectly before).

Begin circuit board layout.

Monday, 6th

Decision made to include temperature sensor on prototype. This would allow the prototype to be used in place of any final design for firmware and software development, and would enable firmware and software to be developed against a similar target if it becomes clear that there is insufficient time to complete and have a final design built.

Temperature sensor selection started, expected to be completed tomorrow.

Tuesday, 7th

The NXP PCT2075TP is selected as the temperature sensor. Circuit symbols for the device for use in EAGLE created.

Wednesday, 8th

General work on circuit board design done.

Thursday, 9th — Week 9

Further general work on circuit board design done. Design largely completed.

The proposed schedule listed this day as the day hardware manufacturing was to begin, and the day by which firmware and software development was to be completed. However, the proposed schedule was made when it was believed that there would be less time available. Since the making of the proposed schedule, it has been clarified that a first draft of the report must be submitted by the 23rd of February 2018 and that the implementation-phase deadline of 22nd December 2017 was a general recommendation and not a rule.

Progress from this point onward will not follow the proposed schedule.

Friday, 10th

Circuit board largely complete, and the manufacturer has provided a quote. The cost, given initially in dollars, is to be around £61.39 for a single prototype, which includes the manufacture of a printed circuit board, the cost of components, and the assembly of the circuit, with a 15-working-day turnaround.

However, in the interest of correctness, final checks will be made before the prototype is sent for manufacture. It is expected that these can be completed tomorrow. If successful, it is likely that the design will be sent for manufacture.

Added to the report are minor notes about the selection of a 20-pin female expansion header for the circuit, and a rationale about the circuit design around the power MOSFETS.

Saturday, 11th

Circuit board design checked, and appears correct. An additional test point was added on the 12 V connection. The design was not sent for manufacture, as the manufacturer does not include weekends in its predicted turnaround time and so ordering today or tomorrow makes no difference. By postponing ordering until tomorrow, the design and every component can be double-checked.

Sunday, 12th

Bill of Materials drawn up and present in Appendix F. The circuit could not be ordered as personal commitments prevented updating the circuit (consequently from the drawing up of the Bill of Materials revealing that adjusting the selection of parts—such as selecting resistors in a different form factor—saved cost) and any double-checking.

Monday, 13th

Circuit updated in consequence of the aforementioned parts changes.

The final cost for the prototype comes to the equivalent of £56.33, with the inclusion of shipping costs bringing the total cost to around £79.82 (as prices were given by the manufacturer in dollars, current exchange rates were used to determine these approximate costs). An order was placed for the prototype at this cost, with 15-working-day turnaround giving a projected delivery date of the 4th of December.

Other minor updates were made to the main report.

Wednesday, 15th

Begin adding circuit schematics, designs to Appendix F.

Thursday, 16th Week 10

Set out basic program specification for proof-of-concept prototype firmware.

Further work on Appendix F.

Friday, 17th

Minor other work on Appendix F, and Appendices F1 to F3 are provisionally complete.

Information about relevant law transferred to main report body from notes.

Saturday, 18th

Begin further work on proof-of-concept prototype firmware design.

Sunday, 19th

Minor firmware design work.

Monday, 20th

Minor firmware design work.

Tuesday, 21st

Proof-of-concept prototype firmware general design provisionally completed, but it is expected that changes will be made as the firmware is produced. Should now be able to progress to writing the firmware, but limited time makes it unlikely that this will begin before Thursday.

Thursday, 23rd Week 11

General familiarisation with the Silicon Labs “Simplicity Studio” IDE, the device to be used, and related. Work on device firmware has begun, with the very basic groundwork for the firmware started.

As the proof-of-concept prototype circuit will not be available before December, it is anticipated that there could be not insignificant rework required to portions of firmware produced before receipt.

Friday, 24th

Further familiarisation work, and minor work on firmware. Notable for future reference is that hardware peripheral interrupts must both be enabled in the registers which control the peripheral, and in the nested vectored interrupt controller (NVIC). If NVIC configuration is not performed, the interrupt service routine will never be executed.

Saturday, 25th

Groundwork of microcontroller interrupts required for testing largely in place, and what is present confirmed to work. Again, complete testing cannot be done at this time due to the proof-of-concept prototype circuit not being available.

Sunday, 26th

Checking the datasheet for the microcontroller on the development kit indicated that neither the pulse-counting or PWM functions were available and connected to pushbutton inputs or LED outputs, respectively, on the development kit. Any code for operating these peripherals cannot be tested before the proof-of-concept prototype arrives.

Monday, 27th

Begin work on summary of expenditure.

Tuesday, 28th

Purchase computer fans to be used in testing. Purchases added to the summary of expenditure.

Thursday, 30th Week 12

Begin drawing up list of interrupt dependencies. As referenced in the entry for Sunday, 26th, there are multiple configuration options which must be adjusted before the microcontroller will generate and service and interrupt. Drawing up a list of these options will hopefully serve to reduce time spent on debugging.

DECEMBER

Friday, 1st

Minor further work on the list of interrupt dependencies.

Saturday, 2nd

List of interrupt dependencies provisionally complete. Begin drawing up list of clock dependencies.

Each (or nearly each) peripheral or hardware device on the microcontroller can be individually enabled or disabled to control energy consumption, and so the relevant clock for that peripheral must be enabled before it can be configured or used. The list of clock dependencies is hoped to reduce development time, as it will act as a quick reference of required clocks and negate the need to continually search through the reference manual.

Sunday, 3rd

List of clock dependencies provisionally complete.

Monday, 4th

Add USB and DMA (direct memory access) to the list of clock dependencies. Use of USB (and likely DMA) isn't required for the basic testing, but the information is useful to have ready for future reference.

Tuesday, 5th

Despite it not being possible to fully test, begin writing firmware to operate peripherals. Whatever is written will need to be simple enough that pinpointing errors (which there are expected to be, given the code is almost entirely untested) is not overly cumbersome.

Thursday, 7th — Week 13

Minor work on test firmware.

Friday, 8th

Minor work on test firmware.

Update the summary of expenditure to remove the postage charge from the entry for the 28th of November. The postage charge was for delivery on the 30th, but delivery of the ordered items was late and, as a result, the charge was refunded.

Saturday, 9th

Add to the test firmware code configuring the development kit LCD. Initially it was believed that it would be required interface with the LCD controller directly, but Silicon Labs provides a driver which includes convenience functions for the pictorial segments and for writing alphanumeric characters to the 14-segment display portions of the LCD.

Sunday, 10th

Update the list of clock dependencies with those for the LCD controller.

Beginnings of a state machine added to the firmware code. Not yet tested as there was insufficient time today. This style of implementation appears to be the easiest method (which is also relatively maintainable) for the firmware.

Monday, 11th

Some testing of the firmware, other minor work done.

Tuesday, 12th

Begin producing a test plan for the proof-of-concept prototype.

Thursday, 14th — Week 14

Add equipment listing to the test plan.

Work on the project is likely to become less regular in the coming two weeks due to both an increased workload and the holiday period.

Friday, 15th

Begin adding action items to the test plan.

Saturday, 16th

Expand list of action items, update equipment listing with descriptions for the components and test points referenced in the listing.

Tuesday, 19th

Pre-emptively add taxes, etc. in relation to the proof-of-concept prototype to the summary of expenditure. As the prototype was manufactured in the United States, VAT and customs duty must be paid on its entry into the United Kingdom. There is also a further fee levied by the delivery service as a result of their having to pay import charges in advance. Were the prototype being manufactured commercially, it would be likely that such charges could be avoided (in the case of manufacture in the United Kingdom).

Wednesday, 20th

Minor firmware refactoring to simplify main loop.

Thursday, 21st Week 15

Received proof-of-concept prototype circuit.

It was expected that the proof-of-concept prototype circuit would be received within a month of ordering. However, as the circuit was instead received a month and one week after ordering, and as additional equipment (power supplies, signal generators, oscilloscopes, etc.) is required to test the circuit, there is insufficient time to test the circuit before the holiday period.

Add to list of action items, and add short notes for each action item explaining its purpose and giving relevant information.

Friday, 22nd

Add to list of action items, notes for action items. Consequential modifications to the equipment listing.

Thursday, 28th Week 16

Friday, 29th

Add to Appendix F5 (firmware design resources) discussion about the firmware design related to measuring fan speed. Next step is to write firmware related to use of the microcontroller’s pulse counters.

Add minor note to the main report introducing Appendices F6 and F7 (test plan and test results).

Saturday, 30th

Initialisation firmware for the pulse counters written. Reading the reference manual in writing the firmware revealed features relevant to the discussion added yesterday. Begin reworking the relevant portions of the discussion.

Among these features is the ability to route (internally, in the microcontroller) to one of the counter's inputs the output of another peripheral via the "peripheral reflex system" (PRS). In particular, the PRS is able to route the signal from a timer firing to the pulse counter, which could enable testing of firmware relying on the pulse counters before connection of the development kit to the proof-of-concept prototype, and so could in turn save valuable testing time.

Sunday, 31st

Complete reworking of discussion. Minor other changes to firmware.

2018

JANUARY

Monday, 1st

Confirmed that using the peripheral reflex system to trigger the pulse counter works. A minor modification from the anticipated configuration was required.

If the pulse counter is in "single input oversampling" mode, the input is sampled on Low Frequency A (LFA) clock pulses, and the LFA clock is 32.768 kHz. This is an issue, as the PRS bus uses the much higher frequency High Frequency Peripheral clock (HFPER), which operates at 14 MHz on first start. The signal transmitted through the PRS is a single HFPER pulse width, and so is seldom registered by the counter.

However, by adjusting the pulse counter to operate in "single input externally-clocked oversampling" mode, where rising edges on the input signal are used to clock the pulse counter, these fast pulses are registered. The impact of this on functions synchronised to a clock must be investigated. Further, the reference manual notes (on page 608) that "the external pin clock source must be configured from the registers in the [Clock Management Unit]." Further changes to configuration may be required for the real-world use-case, where the signal is received on an external pin rather than via an internal bus.

Tuesday, 2nd

Largely firmware housekeeping. Code for firmware states separated so that one state has no access to the other states, and so that the interrupt service routine for the master periodic timer has no access to the internals of any state.

Now that pulse counting code has been confirmed to work, begin implementing code for control mode selection. That is, the code which enables selection of one of the three control mode required by the program specification in the main body of the report.

Thursday, 4th Week 17

Control mode menu code fully working. There are some limitations in what can be done—the LCD on the development kit includes seven 14-segment displays and four 7-segment displays, but the vendor-provided drivers only support the display of numbers (decimal or hexadecimal) on the 7-segment displays. In order to display longer messages, basic text-scrolling functionality was implemented for the 14-segment displays.

To ensure that the current menu option was clear at all times (and not just at a particular point in the scrolling text), the numeric 7-segment displays were used. This required the use of mnemonics which could be constructed from the limited hexadecimal character set accepted by the vendor LCD driver, but it was felt that another method would worsen clarity rather than improve it.

Friday, 5th

Begin writing code for PWM generation. A minor firmware mistake was made in that the low-energy timer (LETIMER0) was configured for use as the master timer, but is used on the proof-of-concept prototype as the PWM control signal driver for the fan connected to the PWM DAC. Correction of this mistake only required the swapping of LETIMER0 for the regular timer TIMER2.

Saturday, 6th

As the operation of the timers in PWM mode is not entirely straightforward, begin writing a note (included as Appendix F5.3 as of today) explaining the general principles and operation for use as a reference. No further PWM code written.

In writing the reference note, a potential design issue was uncovered. The low-energy timer used to provide the fan PWM control signal to the fan connected to the PWM DAC is driven from the Low Frequency A (LFA) clock. The LCD controller is also driven from this clock. At present, the clock is configured with the Low Frequency RC Oscillator (LFRCO) as its source, causing it to operate at a frequency of 32.768 kHz. The potential design issue is that it is not possible to divide 32.768 kHz down to 25 kHz with the microcontroller's clock controls.

However, the LFA clock can also be driven by the HFCORE clock divided by two, which would result in a frequency of 7 MHz in the current configuration. As the LCD controller is also driven from the LFA clock, it may be the case that it is not possible to use the higher-frequency HFCORE in place of LFRCO. This must be investigated. It is not certain why this issue was not identified in design, as a check of the timers was made. The assumption going forward is that, during the design stage, it was expected that LFA would be driven from HFCORE for the LE timer without considering that the LCD controller was also clocked by LFA.

Sunday, 7th

The potential issue uncovered yesterday is resolved. Through practical testing, it was confirmed that operating the LCD controller at a higher frequency does not impact operation in any visible way. The result of this is that the LE timer can be provided with a suitably high frequency without affecting the LCD controller.

To be specific, the LFA clock which drives the LCD controller and LE timer can be configured to be driven from (among others) the 32.768 kHz LFRCO, or from the 7 MHz $\text{HFCORECLK}_{\text{LE}}$. The $\text{HFCORECLK}_{\text{LE}}$ is a derivative of the 14 MHz HFCORE clock, divided by two before driving the LFA clock. To attain as low a frequency as possible, the derivative of it provided to the LCD controller can be further divided by 128 in the `CMU_LFAPRESC0` register (giving 54.6875 kHz), and the LCD framerate derived from this clock can be set up to 4× slower through the register `CMU_LCDCTRL` to give an LCD framerate of approximately 13.672 kHz. A setting to divide HFCORE by 4 is available in `CMU_HFCORECLKDIV`, but in a brief test (using values reported by the `CMU_ClockFreqGet()` function in the "emlib" library provided by the microcontroller vendor) this did not appear to have any effect on the LCD clock.

In this configuration, the LE timer is provided with a 7 MHz clock. Using the LE timer's prescaler in `CMU_LFAPRESC0`, and by configuring the timer to count down from 139, it is possible to produce the 25 kHz required.

Monday, 8th

Complete the note started on Saturday, 6th. The configuration code for the PWM generators is completed. The next step is to write the code for the control modes, including the code necessary to enable and reconfigure the PWM generators as required. Minimal on-the-fly reconfiguration is required—the only such changes would be the state of the generators (enabled or disabled), the duty cycle of the output waveform, and (for the generators required to provide either 100 kHz or 25 kHz depending on the control mode) the clock frequency to the timer.

Tuesday, 9th

Begin writing code for the control modes, including PWM reconfiguration code.

Fix a minor error in the footnote just after the microcontroller connection table. The footnote previously said that the pins "PB12 and PD7 are different output channels for the same PWM function," which is not accurate—they are different output channels for the same timer, but PWM functionality is configured on a per-channel basis (albeit with the limitation that each PWM function outputs a waveform at the same timer-determined frequency).

Wednesday, 10th

Extremely minor additions to control mode code.

Thursday, 11th — Week 18

Photographs of the proof-of-concept prototype and the development kit taken for inclusion in Appendix F4 (as of writing). Minor additions to the discussion on the microcontroller's PWM generators.

It is expected that some testing of the proof-of-concept prototype can be done tomorrow, although the extent is not known.

Friday, 12th

The proof-of-concept prototype was tested. No action items were completed, as the testing performed revealed an issue which requires further investigation. The results of this test, with discussion, was written up in Appendix F7.1.

Saturday, 13th

Further work on the write-up mentioned yesterday, primarily in relation to the discussion of the results and potential future actions.

Sunday, 14th

Firmware code quality improvements—identify some repeated code segments in firmware and separate into utility functions; other minor changes.

Monday, 15th

Firmware control modes code preliminarily complete.

Code for all control modes is largely the same, but some trouble was encountered in separating out the common portions due to the amount of state required. It is possible, but results in functions with many parameters. May still separate common portions out.

Tested basic speed-measurement code using a PRS-connected timer as input to the appropriate pulse counter. The code worked, but appeared to miss one or two pulses—with the pulse-generating timer configured for 40 Hz (2400 pulses per minute, equivalent to 1200 rpm) the code reported 1140 rpm, and with the timer configured for 25 Hz (1500 ppm or 750 rpm) a speed of 690 rpm was reported. This may be an issue that results from the clocks in the microcontroller being synchronised.

An addition to consider is the ability to count more than 255 pulses per period. This is not a critical feature—few fans are likely to generate more than 255 pulses in a given measurement period (as, for example, this would be over 15,000 rpm for a measurement period of a second)—but would be a quality improvement.

Tuesday, 16th

Minor design error discovered while implementing the fan PWM signal control mode. It appears that the definitions for microcontroller pins PB11 and PB12 (11 and 13 on the expansion header) were reversed. Schematics show that expansion header pin 13 is TIMER1_CC2, and that 11 is LETIM0_OUT1, but—in fact—the reverse is true. As both are timer functions capable of producing PWM output, this design error is inconsequential but does require a review of code to ensure that the correct timer is used.

This was not caught during design review before manufacturing as that review incorrectly assumed that the pin definitions were correct, and instead focused on ensuring that the connections for those pin definitions were correct.

Schematic diagrams in Appendix F are updated accordingly.

Thursday, 18th Week 19

Basic further testing of the proof-of-concept prototype, preliminarily confirming that a resistor in the circuit is non-functional and may have been causing the issue which was discovered on Friday, 12th. Summary of the tests and result written up in Appendix F.

Friday, 19th

Further testing of the proof-of-concept prototype. Action items 1, 2, and 3 (as of writing) mostly completed. The source of the issue discovered on Friday, 12th was found. Summary of the work on action item 1 completed, summary of the work on action item 2 started.

Saturday, 20th

Summary of the work on action items 2 and 3 completed. Summary of the work on action item 1 updated regarding potential fixes for the issue from Friday, 12th.

Sunday, 21st

Minor changes made to the summaries of action items 1 to 3. In the summary for action item 3, particular discussion related to the voltage observed at the op-amp outputs was added. The op-amps were unpowered during the preliminary test, and so their output may not have been reliable. Action items 1 and 2 were updated to include a reference to connecting the 3V3 line which powers the op-amps.

Also updated the specific considerations in Appendix F relating to measurement of fan speed. A rolling average was previously discounted but, as the controller would determine when substantial changes in fan speed occur, a rolling average could be a viable method.

It was also determined that the pulse-dropping issue discussed on Monday, 15th may be related to the use of a different pulse counter mode with the PRS.

Monday, 22nd

Begin work on ancillary prototypes. The proof-of-concept prototype tests a key portion of the project, but work must be done in parallel with its testing in order to ensure that the project is completed in time.

Ancillary prototypes are to demonstrate what is required to implement other aspects of the project, such as communication with the host computer over USB.

The first step in this process is to lay out the design requirements for each of the ancillary prototypes. These are not as extensive as those for the proof-of-concept prototype, and deal only with specific portions of the project.

Tuesday, 23rd

Additional work on setting out the requirements, specification, etc. for the USB ancillary prototype. Minor other related work.

Wednesday, 24th

Verified that the development kit appeared to work as intended when powered from a CR2032 button cell.

Thursday, 25th Week 20

Update list of action items to include pin numbers for the pins referenced in items for testing which involves only the development kit.

Provisionally complete the section in the main body of the report on the USB prototype, and begin on Appendix G (ancillary prototypes). Other minor work, largely editing, done on the report main body.

Friday, 26th

Action items 1, 2, and 7 completed. Some work towards action items 3, 4, and 5 done, but there are issues still to be resolved. Provisionally complete summaries for the completed action items, minor work done in relation to the other action items mentioned.

Saturday, 27th

Summaries for action items 3, 4, and 5 provisionally complete.

Begin setting out in Appendix G overview and general considerations for the firmware portion of the USB prototype.

Sunday, 28th

Determined the cause of the issue with action items 3, 4, and 5 that mentioned on Friday, 26th. The issue, that no output was observed on the pins where output was expected, was a result of there being a requirement for output to be enabled both in the configuration for the peripheral (in this case, a timer) and in the configuration for the microcontroller's GPIO. This written up in Appendix F.

Fixes for the abovementioned issue made to the firmware. Whether this is also required for other peripherals which take input (such as the pulse counters) is not known. No mention of input is made in the reference manual, only output. For the time being, no change will be made for input.

Other minor work on Appendix G.

Monday, 29th

Minor work on Appendix G.

Tuesday, 30th

Arranged for access to an electrical workshop to perform further testing.

Add further action items to Appendix F, with notes.

Wednesday, 31st

Update to the summary of findings, adding minor notes which had not previously been applied.

FEBRUARY

Thursday, 1st Week 21

Completed action items 3, 4, 5, and 6. Carried out action items 8 and 9. Further work done in relation to the completed action items 2 and 7. Begin summarising the results of this work in the test results.

Friday, 2nd

Provisionally completed summary for action items 8 and 9. Summary of further work done relating to the completed action item 7 added. Begin on summarising the work carried out in relation to action items 3, 4, and 5.

Saturday, 3rd

Provisionally completed summary for action items 3, 4, 5, and 6.

Sunday, 4th

Minor code quality changes, moving the corrected code for adjusting a timer's prescaler (other than during operation) into a utility function.

Begin setting up USB prototype development environment. No real development work was done as the middleware provided by the microcontroller vendor did not work without modification.

The middleware used a developer-provided C header file for its configuration, which the compiler would report could not be found (if the middleware were included in the recommended manner). To fix this, the middleware source was required to be copied from its default location into the working directory. Here, the compiler then reported that a number of data types were not defined. These types, `uint8_t` (unsigned 8-bit integer) and `uint16_t` (unsigned 16-bit integer) were manually defined, using the definitions provided with the middleware (but which were inactivated by pre-processor directives).

Once these changes were applied, the code—a `main` function empty but for a call to the errata-fixing `CHIP_Init` function provided by the vendor's libraries for the microcontroller—compiled successfully.

Monday, 5th

Add action item 10 to Appendix F. Compile a list (not included in the report) of the steps to be taken for the completion of action items 8 and 9.

Minor additions to USB prototype firmware setup code.

Tuesday, 6th

Make modifications to proof-of-concept firmware needed for action item 10.

Clock configuration code for USB prototype provisionally complete, untested.

Wednesday, 7th

Tested clock configuration code for USB prototype, confirmed to work.

Thursday, 8th — Week 22

Complete action item 10, further work in relation to action items 8 and 9. Begin summarising the work done in Appendix F.

Friday, 9th

Complete additions to summary for action items 8 and 9.

Saturday, 10th

Complete summary for action item 10.

Sunday, 11th

Update proof-of-concept prototype firmware documentation, check that all fixes are applied as necessary, and other minor code quality changes. Add proof-of-concept firmware code to Appendix C and format.

Monday, 12th

Begin in Appendix G remarks relating to the USB prototype. First item added to the remarks is a description of the errors mentioned on Sunday, 4th. Also applied code quality changes to the fix for that issue.

Tuesday, 13th

Add to Appendix G general discussion about the development of firmware for the USB prototype. Research into use of the vendor USB driver, with minor code additions to the firmware. Also confirmed that the Universal Windows Platform library for interfacing with WinUSB was usable in a non-UWP program.

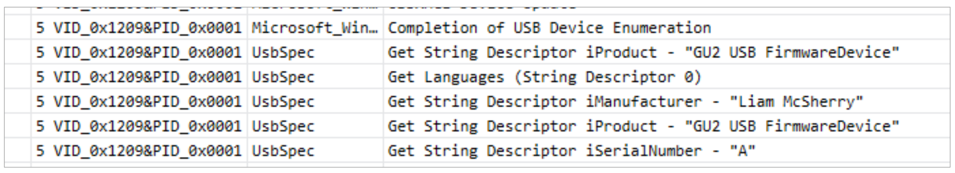
Wednesday, 14th

Significant work towards achieving basic USB configuration. Basic configuration enables verification that the software making use of WinUSB is working, and is required to implement demonstrator firmware emulating the protocol specified in Appendix D.

Thursday, 15th — Week 23

Considering the work remaining, it is not likely that the project will be completed within the recommended 24-week period. However, this was expected from the outset, and work was started early to ensure that the project would be completed before the final deadline on the 20th of April. While it is difficult to predict the exact time required, the current target is to finish work on the project in mid to late March, with the remaining time allocated to the final writing up, editing, and review, and having the report printed.

Basic USB configuration achieved. The development kit was programmed with the firmware and connected to a host computer. The host computer indicated it had acknowledged the device. Device identification was observed using the Microsoft Message Analyser tool (see below).



```
5 VID_0x1209&PID_0x0001 Microsoft_Win… Completion of USB Device Enumeration
5 VID_0x1209&PID_0x0001 UsbSpec        Get String Descriptor iProduct - "GU2 USB FirmwareDevice"
5 VID_0x1209&PID_0x0001 UsbSpec        Get Languages (String Descriptor 0)
5 VID_0x1209&PID_0x0001 UsbSpec        Get String Descriptor iManufacturer - "Liam McSherry"
5 VID_0x1209&PID_0x0001 UsbSpec        Get String Descriptor iProduct - "GU2 USB FirmwareDevice"
5 VID_0x1209&PID_0x0001 UsbSpec        Get String Descriptor iSerialNumber - "A"
```

As discussed in the report, the VID–PID pair 1209:0001 is a pair allocated for use privately and in testing by John Otander (of pid.codes). As can be seen, there was a slight error in that a space was missed in the reported name of the device, but this is cosmetic (and so largely inconsequential).

Friday, 16th

Contacted a printers' and was advised that a typical turnaround of 2 days was available (depending on work volume) with plastic comb binding.

Begin adding discussion to Appendix G relating to the Microsoft OS Descriptors specification's operation. Also confirmed that USB v2.1 operation is required, and made changes to the firmware as necessary to attain basic USB v2.1 configuration.

Saturday, 17th

Minor corrections to Appendix F.

Attained WinUSB configuration, with the operating system loading WinUSB for the device automatically on connection. Not yet successful in producing software which can access the USB device.

Sunday, 18th

Further work towards software access for the WinUSB device. Believed that it is necessary to register a device interface GUID, which is ostensibly possible by providing particular Microsoft-specific USB descriptors. The USB device is set to provide such a descriptor, but Windows reports that the descriptor is invalid. As the cause of its being invalid is not apparent, work continues.

Monday, 19th

Software access to the USB device through WinUSB attained. The operating system was unable to recognise the device until the order of descriptors provided by the device was switched. The reason for this being significant is not clear, but this means that it is now possible to begin implementing the Appendix D protocol.

Tuesday, 20th

Begin adding to Appendix G discussion on the sensors prototype. There does not appear to be sufficient time remaining before the 20th of April to design, produce, and test a sensors prototype, and so what would be done to produce a sensors prototype will be discussed in Appendix G2.

Wednesday, 21st

Further work on Appendix G discussion on the sensors prototype.

Thursday, 22nd Week 24

Minor corrections to Appendix D.

Begin implementing the Appendix D protocol in firmware and software, and add to Appendix G discussion and remarks relating to this implementation.

Further work relating to the sensors prototype.

Friday, 23rd

Format the proposed schedule Gantt chart for use in Appendix A1 and reconcile differences between it and the main report summary, which has not been updated since it was first written (and which, for example, used a 30th March deadline and not the 20th April deadline which was later clarified).

Minor correction to a reference in the main report.

Saturday, 24th

Minor corrections to Appendices D and G.

Implement the `GET_FAN_ID` USB request in firmware and software, untested, and update the discussion for both firmware and software in Appendix G.

Sunday, 25th

Tested the implementation of the `GET_FAN_ID` request, correct an error in the implementation, and update software and firmware discussion in Appendix G.

Begin implementing the `SET_FAN_MODE` request—foundational code written for accepting, processing, and responding to user input while maintaining the status output, and the basics of a command language (for specifying Appendix D mode data) set out. The discussion in Appendix G updated.

Modify Appendix D list of USB requests to specify the `bmRequestType` value in writing rather than giving an opaque binary value.

Monday, 26th

Add to Appendix G sensors prototype discussion on I^2C peripheral initialisation.

Tuesday, 27th

Provisionally complete Appendix G sensors prototype discussion section on I^2C initialisation. Begin working on I^2C communication discussion.

Wednesday, 28th

Provisionally complete I^2C-related discussion for the sensors prototype.

MARCH

Thursday, 1st Week 25

Provisionally complete (and test) the software implementation of `SET_FAN_MODE`, and do minor work towards implementing the firmware portion. Appropriate updates to the discussion in Appendix G made.

Friday, 2nd

Minor corrections to Appendix D.

Provisionally complete (and test) the firmware `SET_FAN_MODE` implementation, and correct defects in the software implementation. Discussion on the topic in Appendix G updated. The next step is implementing `GET_FAN_MODE`.

Saturday, 3rd

Provisionally complete the USB prototype, and update Appendix G discussion on its implementation. Begin and largely complete inserting code into Appendix C.

Sunday, 4th

Finish inserting code into Appendix C. It is not expected that any further changes will need to be made to this appendix, other than adjusting the page numbers for final printing.

Begin sensors prototype discussion relating to current and voltage sensing.

Monday, 5th

Further work on sensors prototype current and voltage sensing.

Tuesday, 6th

Minor work on sensors prototype current and voltage sensing discussion.

Wednesday, 7th

Provisionally complete sensors prototype (and Appendix G) discussion. The next step in the implementation of the project to produce recommendations for a final design based on what was learned from the prototypes. When that is completed, the evaluation phase of the project can begin.

This is in line with the timescale discussed on Thursday, 15th of February.

Thursday, 8th — Week 26

Begin work on a section of the main report making recommendations for a final fan controller design. This is the last general stage before the evaluation phase can begin, and should be completed by mid to late March.

Portions added include minor discussion on the host–controller interface, and the start of discussion on power delivery.

Friday, 9th

Further work on power delivery discussion. Discussion added considers which USB connector or connectors should be used, with a start made on the discussion relating to circuit protection if the Intel-specified internal USB header is used.

Saturday, 10th

Revise some discussion added yesterday. Further work on protection and control in the fan controller, including discussion on specific methods. It is expected that this will be completed tomorrow.

Sunday, 11th

Provisionally complete discussion on power delivery. Begin work on discussion relating to the computer fans (control, monitoring, etc.).

Monday, 12th

Minor work on fan speed monitoring—knowledge gained from testing the proof-of-concept prototype, the selection of simple supply modulation rather than use of the PWM DAC for speed control, and the proposed conditioning for the output of the tachometer.

Tuesday, 13th

Minor further work on fan speed monitoring. Considerable time spent attempting to understand Schmitt trigger datasheet which, on comparison with other similar datasheets, appeared incomplete—the sheet provided the maximum positive-going voltage, the minimum negative-going voltage, and the minimum hysteresis voltage, and no other relevant data. The minimum positive-going and maximum negative-going and hysteresis voltages, for example, were not provided. A device with a complete datasheet was chosen as the recommendation going forward.

Wednesday, 14th

Provisionally complete discussion on fan speed monitoring. Minor work on the discussion on fan speed control.

Thursday, 15th — Week 27

Minor corrections to discussion on fan speed monitoring.

Provisionally complete discussion on fan speed control, fan supply voltage and current monitoring, and fan functionality monitoring. Work on the structuring of remaining production design-related sections to be completed.

Friday, 16th

Minor corrections to Appendix F.

Provisionally complete discussion on form factor.

Saturday, 17th

Begin discussion on the host computer driver stack.

Sunday, 18th

Add note to power delivery discussion on protection against the inductive flyback a fan will produce when disconnected from the supply, and include a prospective short-circuit current estimate for the 12 V supply.

Provisionally complete discussion on the host computer driver stack.

Begin discussion on the host–controller protocol.

Monday, 19th

Minor corrections to Appendix G.

Minor work on host–controller protocol discussion.

Tuesday, 20th

Further work on host–controller protocol discussion. This discussion is likely to be completed tomorrow. Once this is completed, the aim is to add discussion for miscellaneous aspects of a production design, for completion in late March in line with the target set on Thursday, 15th of February.

Wednesday, 21st

Provisionally complete host–controller protocol discussion.

Begin setting out structure of miscellaneous discussion.

Thursday, 22nd — Week 28

Add further suggestion for a specific change to the host–controller protocol.

Discussion on the expansion of available ADC channels and on the upgrading of fan controller firmware added. The former is provisionally complete.

Friday, 23rd

Provisionally complete discussion set out on Wednesday, 21st of March. If no further discussion topics are identified, the project can move to the evaluation phase in line with the schedule set out on Thursday, 15th of February.

Sunday, 25th

Begin on a “Conclusion and Review” partition with a critical evaluation of the aim, objectives, and requirements. The evaluation for the aim completed, with an evaluation of the schedule objective largely completed.

Add to Appendix A2 a summary of revisions made to the proposed schedule.

Monday, 26th

Provisionally complete the evaluation of the schedule objective. The evaluation of the technical knowledge objective provisionally completed.

Tuesday, 27th

Minor corrections to Appendix C.

Add discussion on non-removable storage to the miscellaneous discussion on the production design for a fan controller.

Provisionally complete the evaluation of the documentation objective. Begin on the evaluation of the per-unit cost objective.

Wednesday, 28th

Provisionally complete evaluation of the per-unit cost objective.

Thursday, 29th Week 29

Provisionally complete the evaluation of the development cost objective. A final figure cannot be known until this report is printed but, given that spending before printing equals £133.54, it can be said with certainty that the objective is met. The final figure can be added to the evaluation once it is known.

Provisionally complete the evaluation of the test suite objective.

Begin evaluation of the standards compliance objective.

Friday, 30th

Provisionally complete evaluation of the standards compliance objective, largely complete the evaluation of the design for manufacturing objective.

Saturday, 31st

Provisionally complete the evaluation of the design for manufacturing objective.

Begin evaluation of the hardware requirements.

APRIL

Sunday, 1st

Provisionally complete evaluation of the hardware and firmware requirements.

Monday, 2nd

Provisionally complete evaluation of the software requirements.

Add and format references, list of figures, and list of tables.

Tuesday, 3rd

Add tables of contents for each partition, minor editing work.

Wednesday, 4th

Provisionally complete abstract. Begin preparing for print.

Thursday, 5th Week 30

Review and proofread the main body of the report to chapter 13.

Delete Figure 4, the related discussion, and Appendix C1. These portions of the report added nothing and were partially incorrect. Minor style-related and other changes made at numerous points as part of the review and proofreading.

Friday, 6th

Review, proofread, and edit up to chapter 16.4.

Saturday, 7th

Complete proofreading, etc. of the main body of the report.

It is considered that Appendix B (this appendix) does not require review, as there is little discussion, and the discussion present is written in an abbreviated form.

Review and proofread Appendix C. The source code in Appendix C has not been proofread or reviewed, as this was done when it was added to the appendix.

Review and proofread Appendix D.

Appendix E is not considered to require review, as its content is largely a set of tables and not discussion.

Review, proofread, and edit Appendix F to Appendix F7.6.

Sunday, 8th

Complete proofreading, etc. of Appendix F and Appendix G.

Assemble document for print and request quotes from a number of printers.

Monday, 9th

Report sent for printing, and Appendix E and the evaluation of the development cost objective are updated to reflect the cost of printing.

• PAGE INTENTIONALLY LEFT BLANK •

Appendix C
Source Code

C1 LC Filter Underdamped Response Plot

The following is the source code for the script, written in R, used to plot the figure showing the response of the PWM DAC's LC filter when a 400-ohm load is connected, as shown in Figure 6 on page 34.

```r
# Data exported from SPICE in two columns:
# 1   Time
# 2   Voltage
Data <- read.table("Waveform-400R-Load-No-Snub", header=TRUE)
Time <- Data[,1]
V    <- Data[,2]

# Seconds to milliseconds
Time <- Time * 1000

# Plot voltage with grid
plot(Time, V, type="l",
     xlab="Time (ms)", xlim=c(0, 12),
     ylab="Filter Voltage (V)", ylim=c(0, 30),
     lab=c(6, 6, 7), lwd=2)

grid()

# Mark maximum
abline(h=24.34, v=NULL, col="red")
points(x=0.14868, y=24.34, col="red", pch=16)
text(x=0.3, y=26, "24.4 V", col="red")

# Mark supply
abline(h=12.6, v=NULL, col="blue")
arrows(x0=10, y0=8, x1=10, y1=12.6,
      length=0.125, lwd=2, col="blue")
text(x=10, y=6.2, "12.6 V\n(Supply)", col="blue")

# Mark difference
arrows(x0=7.09, y0=12.6, x1=7.09, y1=24.34,
       code=3, length=0.125, lwd=2)
text(x=7.65, y=18.4, "11.8 V")

# Title etc.
title("LC Filter Response at 12.6V with 400-ohm Load")
```

C2 Proof-of-Concept Prototype Firmware

The following is the source code for the proof-of-concept prototype firmware, written in C. The firmware controls the ARM Cortex-M4F microcontroller on a Silicon Labs EFM32WG-STK3800 development kit to operate the prototype as required to carry out the tests in Appendix F6.

main.c

The firmware entry point, and general microcontroller configuration.

```c
/* Author: Liam McSherry
 * Date:   11th February 2018
 * Notes:  Entry point and configuration. Initial chip and
 *         peripheral config. is done in this file.
*/

// Standard C headers
#include <stdbool.h>
#include <stdint.h>
#include <limits.h>
// Device-specific headers for the EFM32WG990-F256 chip.
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_timer.h"
#include "em_pcnt.h"
#include "em_letimer.h"
#include "em_prs.h"
#include "segmentlcd.h"
// Functions etc. from the development kit board support package.
#include "bsp.h"
// Firmware headers
#include "config.h"
#include "states.h"

// Local function prototypes
void configure_Clocks(void);
void configure_masterTimer(void);
void configure_GPIO(void);
void configure_LCD(void);
void configure_PCNT(void);
void configure_emulatedFan(void);
void configure_PWM(void);

// Entry point
int main(void)
{
    // Initialisation, erratum fixes, calibration, etc.
    CHIP_Init();

    // Microcontroller set-up.
    configure_Clocks();
    configure_GPIO();
    configure_LCD();
    configure_PCNT();
    configure_emulatedFan();
    configure_masterTimer();
    configure_PWM();

    /* Infinite loop */
    while (1) {
    }
}

// Configures the clocks for the chip.
void configure_Clocks(void) {
    // Clocks configured are those set out in chapter 14.4 of
    // the main report, and are those clocks required for all
    // peripherals listed (except USB).
    //
    // The processor core clock (HFCORE) is driven by HFRCO at
    // 14 MHz on start, per manual section 11.3.2. We therefore
    // don't need to enable this oscillator or do any
    // configuration for it.
    //
    // The low-energy derivative of HFCORE, called HFCORECLKLE, is
    // initially set to be HFCORE divided by 2, but the divisor
    // can be set to 4.
    //
    // ULFRCO is fixed at 1 kHz.

    // Configure Low-Frequency Clock A (LFA) to use HFCORECLKLE.
    CMU_ClockSelectSet(cmuClock_LFA, cmuSelect_HFCLKLE);


    // Enable clocks for timers 0 to 3
    CMU_ClockEnable(cmuClock_TIMER0, true);
    CMU_ClockEnable(cmuClock_TIMER1, true);
    CMU_ClockEnable(cmuClock_TIMER2, true);
    CMU_ClockEnable(cmuClock_TIMER3, true);

    // Enable clock for low-energy peripheral access
    CMU_ClockEnable(cmuClock_CORELE, true);

    // Enable clock for low-energy timer 0
    CMU_ClockEnable(cmuClock_LETIMER0, true);

    // Enable clock for analogue-to-digital converter 0
    CMU_ClockEnable(cmuClock_ADC0, true);

    // Enable clocks for pulse counters 0 and 2
    CMU_ClockEnable(cmuClock_PCNT0, true);
    CMU_ClockEnable(cmuClock_PCNT2, true);

    // Enable clock for I2C bus 1
    CMU_ClockEnable(cmuClock_I2C1, true);

    // Enable clock for general-purpose I/O
    CMU_ClockEnable(cmuClock_GPIO, true);

    // Enable clock for direct memory access
    CMU_ClockEnable(cmuClock_DMA, true);

    // Enable clock for PRS use.
    CMU_ClockEnable(cmuClock_PRS, true);

    // The manual instructs that changes to these values should be
    // made before the clock for the LCD controller is enabled (or
    // while it's disabled). If this were not this case, this
    // additional configuration would be in the [configure_LCD]
    // function.
    //
    // Divide HFCORECLKE by 128 (LCD controller frequency is
    // ~54.6875 kHz).
    CMU_ClockDivSet(cmuClock_LCDpre, cmuClkDiv_128);
    //CMU->LFAPRESC0 |= CMU_LFAPRESC0_LCD_DIV128;
    // The LCD frame-rate is the controller frequency divided by
    // x+1, so x = 3 -> F.R. == 54.6875 kHz/4 == 13.672 kHz.
    CMU_LCDClkFDIVSet(3);
    // Clock enables
    CMU_ClockEnable(cmuClock_LCDpre, true);
    CMU_ClockEnable(cmuClock_LCD, true);
}

// Configure TIMER2 as 10ms periodic timer.
//
// TIMER0/1 and LETIMER0 are used to drive the fans, and TIMER3 is
// used to emulate a fan in testing, so TIMER2 must be used for
// master timer. Although, it would be possible to swap the uses
// of TIMER2/3, it wouldn’t make any real code difference.
//
// The SysTick timer included with the ARM processor in the
// microcontroller could be used, but it isn't fixed frequency.
// Instead, firmware is required to use a value (indicating the
// number of ticks per 10ms) to calculate the required interval as
// a number of ticks. Using TIMER2 with a known and fixed
// frequency, the need for calculation is largely removed. The
// downside is that this particular timer won't be guaranteed
// portable between ARM processors, but that's highly unlikely to
// be an issue.
void configure_masterTimer(void) {
    // The timer is driven from the High-Frequency Peripheral
    // Clock (HFPER), which pulses at 14MHz when the
    // microcontroller first starts. The timer has a prescaler and
    // can be configured to only fire once it has counted a
    // certain number of times. If the timer is configured to
    // count up, its firing period is given by the following
    // formula:
    //
    // Time = (1/HFPER) * Prescale * (Period_Adjust + 1)
    //
    // To achieve the 10ms master timer period, the following
    // values are used:
    //       Prescale      = 8
    //       Period_Adjust = 17499 (0x445B)
    //
    // These values are reflected in the configuration.

    // The bulk of the initialisation data for the timer.
    const TIMER_Init_TypeDef init = {
        // The timer is disabled and not counting after
        // initialisation.
        .enable        = false,
        // The timer does not continue to count in debug mode.
        .debugRun      = false,
        // Divide the timer-driving clock by 8
        .prescale      = timerPrescale8,
        // The timer is driven from HFPER
        .clkSel        = timerClkSelHFPerClk,
        // The timer does not count at 2x counting rate
        .count2x      = false,
        // Don't always track input polarity (since we aren't
        // using inputs)
        .ati          = false,
        // Take no action on falling/rising input edges (since we
        // aren't using the inputs anyway)
        .fallAction   = timerInputActionNone,
        .riseAction   = timerInputActionNone,
        // Don't clear DMA requests for the timer
        .dmaClrAct    = false,
        // The timer counts up to the Period_Adjust value
        .mode         = timerModeUp,
        // We aren't using the quadrature decoder, so this doesn't
        // matter
        .quadModeX4   = false,
        // The timer counts continuously until stopped by software
        .oneShot      = false,
        // The timer is not stopped/started/reloaded by other
        // timers
        .sync         = false
    };

```

```c
    // We're using TIMER2 as TIMER0/1 and LETIMER0 are used to
    // drive the fans, and TIMER3 is used to provide an emulated
    // fan tachometer signal during testing.
    TIMER_Init(TIMER2, &init);

    // Enable interrupt generation and handling
    TIMER_IntEnable(TIMER2, TIMER_IF_OF);
    NVIC_EnableIRQ(TIMER2_IRQn);

    // Set the period adjust value
    TIMER_TopSet(TIMER2, 0x445B);

    // Start the timer
    TIMER_Enable(TIMER2, true);
}

// Configures GPIO for:
//  o using the development kid LEDs (PE2 and PE3);
//  o receiving interrupts from the first pushbutton on the
//    development kit (PB9).
//
// Note that on-the-fly GPIO configuration used with the fan
// control modes is done in the state handlers for those states,
// and not by this function.
void configure_GPIO(void) {
    // Port E configured for low-current drive
    GPIO_DriveModeSet(gpioPortE, gpioDriveModeLowest);
    // Pins PE2..3 configured for push-pull output at drive
    // strength. These pins are connected to the LEDs on the
    // development kit.
    GPIO_PinModeSet(gpioPortE, 2, gpioModePushPullDrive, 0);
    GPIO_PinModeSet(gpioPortE, 3, gpioModePushPullDrive, 0);


    // Set up the interrupts for PB9 and PB10, which are connected
    // to the pushbuttons on the development kit.
    GPIO_ExtIntConfig(
        gpioPortB, // Port B is the source
        9,         // On Port B, pin 9 is the input
        9,         // External interrupt 9 (hard-wired, ignored)
        false,     // Don't trigger on the rising edge
        true,      // Trigger on the falling edge
        true       // Enabled
    );

    GPIO_ExtIntConfig(gpioPortB, 10, 10, false, true, true);

    // Pins PB9, 10 to input with filter enabled (to debounce the
    // buttons)
    GPIO_PinModeSet(gpioPortB,  9, gpioModeInput, true);
    GPIO_PinModeSet(gpioPortB, 10, gpioModeInput, true);

    // Clear all GPIO interrupts
    GPIO_IntClear(0xFFFFFFFFU);
}

// Configures the 160-segment LCD on the development kit.
void configure_LCD(void) {
    const LCD_Init_TypeDef init = {
        // LCD controller is enabled after configuration
        .enable    = true,
        // LCD on the development kit has 8 commons, so needs
        // 8-way multiplexing if all segments are to be used.
        .mux       = lcdMuxOctaplex,
        // The bias level for the LCD: the number of distinct
        // voltage steps in the common voltage provided to the
        // LCD. Per AN0057, LCDs with 5 to 8 multiplexed common
        // lines require 1/4 bias. I don't pretend to fully
        // understand this.
        .bias      = lcdBiasOneFourth,
        // Normal wave output (other option is low-power).
        .wave      = lcdWaveNormal,
        // LCD voltage is the microcontroller supply voltage
        .vlcd      = lcdVLCDSelVDD,
        // Contrast is adjusted relative to the LCD common
        // voltage.
        .contrast  = lcdConConfVLCD,
    };

    LCD_Init(&init);

    // Enables all segments for use
    LCD_SEGMENTS_ENABLE();

    // Start with LEDs all off
    SegmentLCD_AllOff();
}

// Configures the pulse counter (PCNT) to track the number of
// pulses
void configure_PCNT(void) {
    // See Appendix F5.2 for considerations specific to fan speed
    // measurement.

    // Due to the way the PRS works, the sampling mode must be
    // changed if it is used to emulate a tachometer. PRS pulses
    // are only a single clock pulse long, and so synchronising
    // the counters with a clock could result in the counter
    // sampling entirely missing the single pulse.
    //
    // Instead, when the PRS is in use, the counters are clocked
    // by their input. This results in some pulses being dropped,
    // but otherwise correct operation is maintained.
#ifdef Config_FanEmu_PrsIncludes
    const PCNT_Mode_TypeDef pcntMode = pcntModeExtSingle;
#else
    const PCNT_Mode_TypeDef pcntMode = pcntModeOvsSingle;
#endif

    const PCNT_Init_TypeDef init = {
        // How the pulse counter is samples input (see above).
        .mode           = pcntMode,
        // The counter value starts at zero.
        .counter        = 0,
        // The top counter value is the maximum that can be stored
        // in an unsigned 8-bit integer (the greatest value
        // capable of being stored by all pulse counters).
        .top            = 255,
        // The counter monitors for positive pulse edges. This
        // hopefully will prevent the issue referred to in
        // Appendix F5.2.
        .negEdge        = false,
        // The counter counts up.
        .countDown      = false,
        // No requirement for a pulse to last 5 clock cycles.
        .filter         = false,
        // Over/underflow to TOP/2 is disabled.
        .hyst           = false,
        // The counter's S1 input does not determine counting
        // direction.
        .s1CntDir       = false,
        // The main counter only counts on up-count events. Since
        // we aren't using the quadrature decoding mode, this only
        // serves to ensure that any potential (and unlikely)
        // down-count events are ignored, and that up-count events
        // are processed.
        .cntEvent       = pcntCntEventUp,
        // The auxiliary counter does not respond to count events.
        .auxCntEvent    = pcntCntEventNone,
        // The peripheral response system (PRS) channels to be
        // used for the counter's S0 and S1 inputs. We don't want
        // to use these, but we have to provide a value.
        .s0PRS          = pcntPRSCh0,
        .s1PRS          = pcntPRSCh0
    };

    // Route GPIOs to the pulse counter inputs.
    PCNT0->ROUTE = PCNT_ROUTE_LOCATION_LOC3; // PD6
    PCNT2->ROUTE = PCNT_ROUTE_LOCATION_LOC0; // PD0
    // Configure GPIOs to be used as inputs to the counters. These
    // must be set as inputs or no signal is transmitted to the
    // counters. Configured to use the controller's internal pull-
    // down resistors.
    //
    // PD6 (PCNT0)
    GPIO_PinModeSet(gpioPortD, 6, gpioModeInputPull, 0);
    // PD0 (PCNT2)
    GPIO_PinModeSet(gpioPortD, 0, gpioModeInputPull, 0);

    // Configure PCNT0 and PCNT2, the counters used in the
    // prototype, as above. PRS inputs are disabled by default.
    PCNT_Init(PCNT0, &init);
    PCNT_Init(PCNT2, &init);
}

// Configuration related to testing the pulse counters. Generally,
// the setup required to internally (within the controller)
// generate a pulse that can be registered by the pulse counters.
//
// Configures the Peripheral Reflex System for signal transmission
// from Timer 3 to the pulse counters (i.e. connect Timer 3's
// overflow signal to channel 0, which the pulse counters are
// configured to listen on).
//
// Configures Timer 3 to initially pulse at 2,400 ppm (40 Hz).
void configure_emulatedFan(void) {
    // For emulating a fan tachometer, we want to use the PRS to
    // transmit a signal from a timer to the pulse counters. We're
    // going to use TMR3, since we'll be using TMR0 and TMR1 for
    // PWM output and we don't want to accidentally have one
    // interfere with the other.
    PRS_SourceSignalSet(
        // Route signal to PRS channel 0
        0,
        // Signal source is TIMER3
        PRS_CH_CTRL_SOURCESEL_TIMER3,
        // Use the TIMER3 Overflow signal as a trigger
        PRS_CH_CTRL_SIGSEL_TIMER3OF,
        // Trigger on the positive signal edge
        prsEdgePos
    );

    // The timer firing period is given by the following equation
    // (for a timer configured in up-count mode):
    //
    //     Time = (1/HFPER) * Prescale * Period_Adjust
    //
    // Where:
    //     HFPER is 14 MHz
    //     Prescale is given by configuration
    //     Period_Adjust is given by configuration
    //
    // To achieve 40 Hz (25 milliseconds), the following values
    // are to be used:
```

```
    //     Prescale      = 16
    //     Period_Adjust = 21875 (0x5573)
    const TIMER_Init_TypeDef init = {
        // Timer is disabled and not counting after
        // configuration
        .enable         = false,
        // Timer does not continue to count in debug mode
        .debugRun       = false,
        // Divide the driving clock by 8
        .prescale       = timerPrescale16,
        // Timer is clocked by the High Frequency Peripheral
        // Clock, which runs at the same 14MHz as HFCORE
        // (produced by HFRCO) unless configured
        // otherwise (which we aren't going to do).
        .clkSel         = timerClkSelHFPerClk,
        // Don't count at the 2x rate
        .count2x        = false,
        // Don't always track input polarity
        .ati            = false,
        // Do nothing on falling/rising input edges (since we
        // don't use inputs)
        .fallAction     = timerInputActionNone,
        .riseAction     = timerInputActionNone,
        // Don't clear DMA requests
        .dmaClrAct      = false,
        // Timer counts up to the Period_Adjust value
        .mode           = timerModeUp,
        // We aren't using the quadrature decoder, so this
        // value doesn't matter
        .quadModeX4     = false,
        // Timer counts continuously (not just once)
        .oneShot        = false,
        // Timer is not stopped/started/reloaded by other timers
        .sync           = false
    };

    // Initialise Timer 3
    TIMER_Init(TIMER3, &init);

    // Set the Period_Adjust value
    TIMER_TopSet(TIMER3, 0x5573);

    // Start the timer
    TIMER_Enable(TIMER3, true);
}

```

```c
// Configures the timers for PWM fan control
void configure_PWM(void) {
    // PWM is provided by the microcontroller's timers. For
    // reference, the timers are used for the following functions:
    //
    //      TIMER0     CC1    Fan (PWM DAC), PWM DAC input
    //      ----------------------------------------------------
    //      TIMER1     CC1    Fan (Bare), fan ground
    //                 CC2    Fan (Bare), PWM control signal
    //      ----------------------------------------------------
    //      LETIMER0   OUT1   Fan (PWM DAC), PWM control signal
    //
    // The shared use of TIMER1 isn't a problem, as no situation
    // has been foreseen that would require modulating the fan
    // supply and providing a standard PWM control signal at the
    // same time.

    // The formula is the same used in [configure_masterTimer],
    // except that the target period is 10 microseconds. This
    // means that the timers will default to the 100kHz PWM
    // frequency required for the PWM DAC input and supply
    // modulation, but also enables easy reconfiguration to the
    // 25kHz frequency required for the PWM control signal.
    //
    // In this case, using the 14MHz HFPER signal, a prescaler
    // value of 1 and a Period_Adjust value of 139 is required to
    // produce 100kHz. To produce 25kHz from this, the prescaler
    // can be adjusted to 4.
    //
    // This configuration is effectively equivalent to that used
    // for the master timer. PWM is configured using the
    // Capture/Compare (CC) channels, rather than the timer
    // itself.
    const TIMER_Init_TypeDef timer_init = {
        .enable     = false,
        .debugRun   = false,
        .prescale   = timerPrescale4,
        .clkSel     = timerClkSelHFPerClk,
        .count2x    = false,
        .ati        = false,
        .fallAction = timerInputActionNone,
        .riseAction = timerInputActionNone,
        .dmaClrAct  = false,
        .mode       = timerModeUp,
        .quadModeX4 = false,
        .oneShot    = false,
        .sync       = false
    };
    // The configuration for the low-energy timer (LETIMER0) is
    // slightly different by reason of its different interface,
    // but the effect is equivalent.
    //
    // Importantly, LETIMER0 would normally be driven by the
    // 32.768kHz LFRCO, but this would make it impossible to
    // produce the 21-28kHz output needed for fan control.
    // Instead, LETIMER0 is driven by the 7MHz HFCORECLKLE (a
    // division of HFCORECLK by 2). As with the normal timers, a
    // TOP value of 139 is used, but this time a prescaler of 2
    // (rather than 4) is required to produce 25kHz.
    const LETIMER_Init_TypeDef letimer_init = {
        .enable         = false,
        .debugRun       = false,
        // Don't start counting when the real-time clock (RTC)
        // matches values in either COMP0 or COMP1
        .rtcComp0Enable = false,
        .rtcComp1Enable = false,
        // When the timer counter underflows, reload the value of
        // COMP0 into the TOP register (i.e. the value to count
        // down from).
        .comp0Top       = true,
        // When the repeat counter REP0 reaches zero, do not load
        // COMP1 into COMP0.
        .bufTop         = false,
        // Outputs are low when the timer is in the idle state
        .out0Pol        = 0,
        .out1Pol        = 0,
        // When the timer underflows while the repeat counter is
        // non-zero, do nothing on output zero.
        .ufoa0          = letimerUFOANone,
        // When the timer underflows while the repeat counter is
        // non-zero, become active when the timer matches COMP1.
        // Regardless of repeat counter state, return to idle on
        // underflow.
        .ufoa1          = letimerUFOAPwm,
        // The timer continues to count until stopped by software.
        .repMode        = letimerRepeatFree
    };

    // Timer initialisation with above values
    TIMER_Init(TIMER0, &timer_init);
    TIMER_Init(TIMER1, &timer_init);
    LETIMER_Init(LETIMER0, &letimer_init);

    // Configure timers for PWM
    //
    // For the regular timers, this is configured on a per-CC-
    // -channel basis.
    const TIMER_InitCC_TypeDef timer_cc_init = {
        // Settings for input, irrelevant as we're using the
        // channel for output. Disregard these values.
        .eventCtrl      = timerEventEveryEdge,
        .edge           = timerEdgeBoth,
        .prsSel         = timerPRSSELCh0,
        // Do nothing when the timer counter under- or overflows.
        .cufoa          = timerOutputActionNone,
        .cofoa          = timerOutputActionNone,
        // When the timer counter matches the CC channel value
        // (stored in the register TIMERn_CCx_CCV), toggle the
        // state of the output.
        //
        // The effect of this is that modifying the channel value
        // modifies the duty cycle of the output PWM waveform.
        .cmoa           = timerOutputActionToggle,
        // The CC channel is in PWM mode. As far as I can tell,
        // the bulk of this mode is (1) the channel value is
        // buffered to avoid any glitches in output, and (2) the
        // output is returned to the high state each time the
        // timer overflows.
        .mode           = timerCCModePWM,
        // Digital filter on inputs. As we're outputting, not
        // required.
        .filter         = false,
        // We don't want to take input from a PRS channel, either.
        .prsInput       = false,
        // In PWM mode, determines the state of the output
        // immediately on being enabled.
        .coist          = false,
        // The output is not inverted.
        .outInvert      = false
    };
    // And we'll be using the same configuration for each channel.
    TIMER_InitCC(TIMER0, /* CC */ 1, &timer_cc_init);
    TIMER_InitCC(TIMER1, 1, &timer_cc_init);
    TIMER_InitCC(TIMER1, 2, &timer_cc_init);

    // Fortunately, we can set the register values once.
    // Configuring the timers for 100kHz allows simple changing to
    // 25kHz by adjusting the prescaler.
    //
    // Regular timers are to be divided by 1 for 100kHz, and the
    // low-energy timer by to 2 for 25kHz. Note that the LE timer
    // is only used for 25kHz operation, and so is configured to
    // 25kHz immediately.
    _util_TIMER_PrescalerSet(TIMER0, 1);
    _util_TIMER_PrescalerSet(TIMER0, 1);
    CMU_ClockDivSet(cmuClock_LETIMER0, cmuClkDiv_2);
    // In all cases, the counter "top" value is 139. For the LE
    // timer, the top value is given by the COMP0 register value.
    TIMER_TopSet(TIMER0, Config_PWM_TimerCompareValue);
    TIMER_TopSet(TIMER1, Config_PWM_TimerCompareValue);
    LETIMER_CompareSet(LETIMER0, 0, Config_PWM_LETimerCompareValue);
    // Set the duty cycle to zero in all cases. It doesn't appear
    // particularly relevant whether the duty cycle starts at 0%
```

```c
    // or 100%, but starting at 0% does mean that no PWM signal
    // will be provided (and hence, portions of the circuit won't
    // be energised) without microcontroller intervention.
    //
    // For the regular timers, the duty cycle is controlled by the
    // channel value. For the LETIMER, it is controlled by COMP1.
    TIMER_CompareSet(TIMER0, 1, 0); // TIMER0 CC1
    TIMER_CompareSet(TIMER1, 1, 0); // TIMER1 CC1
    TIMER_CompareSet(TIMER1, 2, 0); // TIMER1 CC2
    LETIMER_CompareSet(LETIMER0, 1, 0);
    // In addition, the low-energy timer will only generate PWM
    // output if both its repeat counter registers are non-zero.
    // The manual does not specify that these are decremented at
    // all, so a simple value of 1 should suffice.
    LETIMER_RepeatSet(LETIMER0, 0, 1); // LETIMER0 REP0
    LETIMER_RepeatSet(LETIMER0, 1, 1); // LETIMER0 REP1

    // Output must also be enabled for the GPIOs, otherwise the
    // PWM signal will not appear on the microcontroller's pins.
    // Configured for standard (6mA) drive strength, defaulted to
    // the low state.
    GPIO_PinModeSet(gpioPortB, 11, gpioModePushPull, 0);
    GPIO_PinModeSet(gpioPortB, 12, gpioModePushPull, 0);
    GPIO_PinModeSet(gpioPortC,  0, gpioModePushPull, 0);
    GPIO_PinModeSet(gpioPortD,  7, gpioModePushPull, 0);
}
```

irq.c

The code for handling peripheral interrupt requests.

```c
/* Author: Liam McSherry
 * Date:   15th January 2018
 * Notes:  Interrupt request (IRQ) handlers.
*/

// Standard C headers
#include <stdint.h>
#include <stdbool.h>
// Device-specific headers for the EFM32WG990-F256
#include "em_gpio.h"
#include "em_timer.h"
#include "em_letimer.h"
#include "em_pcnt.h"
#include "segmentlcd.h"
// Firmware headers
#include "states.h"

/* Functions named [*_IRQHandler] are declared by the vendor-
 * provided code for use with the microcontroller, and so
 * providing definitions is all that is required here.
 */
```

```c

// Interrupt handler for timer 2 (master timer)'s interrupts.
void TIMER2_IRQHandler(void) {
    // The current state we're in.
    static enum State state = stateIdle;


    // Clear the master timer interrupt flag.
    TIMER_IntClear(TIMER2, TIMER_IF_OF);

    // GPIO interrupt flags
    const uint32_t GPIO_IF = GPIO_IntGet();


    // If we're idling...
    if (state == stateIdle) {
        state = StateHandler_Idle(GPIO_IF);
    }
    // If we're waiting for a control mode to be selected...
    else if (state == stateMenu) {
        state = StateHandler_Menu(GPIO_IF);
    }
    // If we're in one of the control modes...
    else if (state == stateModePwmDac) {
        state = StateHandler_ModePwmDac(GPIO_IF);
    }
    else if (state == stateModeSupplyModulate) {
        state = StateHandler_ModeSupplyModulate(GPIO_IF);
    }
    else if (state == stateModeFanPwmControl) {
        state = StateHandler_ModeFanPwmControl(GPIO_IF);
    }

    // Clear any GPIO interrupts
    GPIO_IntClear(0xFFFFFFFFU);
}
```

config.h

A header containing general configuration-related definitions.

```c
/* Author: Liam McSherry
 * Date:   8th February 2018
 * Notes:  General configuration data and flags.
*/

#ifndef SRC_CONFIG_H_
#define SRC_CONFIG_H_

// The values loaded into the compare registers of the timers.
#define Config_PWM_TimerCompareValue   (139)
#define Config_PWM_LETimerCompareValue (139)
```

```c


// The values to be placed in TIMER3's TOP register to set the
// rate at which pulses are generated for fan emulation.
#define Config_FanEmu_4000rpm     (6562)  // 8000ppm
#define Config_FanEmu_2400rpm     (10936) // 4800ppm (60% 8000ppm)
#define Config_FanEmu_1200rpm     (21875) // 2400ppm
#define Config_FanEmu_750rpm      (36457) // 1440ppm (60% 2400ppm)
#define Config_FanEmu_600rpm      (43749) // 1200ppm
#define Config_FanEmu_420rpm      (62499) //  840ppm (70% 1200ppm)

// The timer used to produce the emulated tachometer output.
#define Config_FanEmu_Timer       (TIMER3)

// Defining causes the inclusion of fan emulation PRS code where
// appropriate.
#undef Config_FanEmu_PrsIncludes

#endif /* SRC_CONFIG_H_ */
```

states.h

Provides definitions required for interfacing with code in `states.c`.

```c
/* Author: Liam McSherry
 * Date:   1st February 2018
 * Notes:  Definitions for states.c.
*/

#ifndef SRC_STATES_H_
#define SRC_STATES_H_

#include <stdbool.h>

// Current firmware state
enum State {
    // Doing nothing, not yet received input.
    stateIdle,

    // Selecting control modes.
    //
    // The program specification in section 14.4 of the report
    // requires that three modes be supported, and this state is
    // to be used to select the desired mode.
    stateMenu,

    // Each of the control modes:
    // - PWM DAC; speed control by lowering the absolute voltage
    //    supplied to the fan.
    // - Supply modulation; speed control by lowering the average
    //    voltage supplied to the fan through pulse width
    //    modulation.
    // - PWM control; speed control using the PWM control signal
    //    line to the fan.
    stateModePwmDac,
    stateModeSupplyModulate,
    stateModeFanPwmControl
};

enum State StateHandler_Idle(const uint32_t);
enum State StateHandler_Menu(const uint32_t);
enum State StateHandler_ModePwmDac(const uint32_t);
enum State StateHandler_ModeSupplyModulate(const uint32_t);
enum State StateHandler_ModeFanPwmControl(const uint32_t);

bool _util_TIMER_PrescalerSet(TIMER_TypeDef*, const uint8_t);

#endif /* SRC_STATES_H_ */
```

states.c

Contains the code for each of the states the firmware can be in, along with utilities relevant to the states.

```
/* Author: Liam McSherry
 * Date:   11th February 2018
 * Notes:  State-handling code, plus utilities, for handling all
 *         states.
*/

// Standard C headers
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
// Device-specific headers for the EFM32WG990-F256
#include "em_cmu.h"
#include "em_timer.h"
#include "em_letimer.h"
#include "em_gpio.h"
#include "em_pcnt.h"
#include "segmentlcd.h"
// Firmware headers
#include "states.h"
#include "config.h"

// The bits representing the first and second pushbuttons on the
// development kit in the GPIO interrupt flags register.
#define PUSHBUTTON_0 (1 << 9)
#define PUSHBUTTON_1 (1 << 10)

// Helper function to scroll text across the LCD.
//
// Parameters:
//     text   = The text to display. May be longer than the 7
```

```c
//              characters that the display would otherwise
//              support displaying.
//     offset = The offset into the string. Increase to proceed
//              through the string. If the offset is greater than
//              the string length, it is wrapped around to zero.
void _util_ScrollText(const char* text, uint32_t offset);
// Helper function for displaying a pulse count as RPM on the
// numeric LCD.
//
// Parameters:
//  pcnt            = The pulse counter the count of which is to
//                    be used.
//  samplesPerMin   = The number of times per minute this function
//                    is used to sample the pulse count.
void _util_DisplayRPM(PCNT_TypeDef* pcnt, const uint8_t samplesPerMin);


// Function executed on each tick in the idle state
//
// Parameters:
//     GPIO_IF            = GPIO interrupt state
enum State StateHandler_Idle(const uint32_t GPIO_IF) {
    // Whether this is the first run.
    static bool firstRun = true;

    // The state we'll report on return.
    enum State state = stateIdle;

    // If this is the first run, display "IDLE" on the LCD.
    if (firstRun) {
        SegmentLCD_Write("IDLE");
        firstRun = false;
    }

    // If the either pushbutton is pressed, we want to move to the
    // menu state.
    if (GPIO_IF & (PUSHBUTTON_0 | PUSHBUTTON_1)) {
        // Reset the first run indicator
        firstRun = true;
        // Transition to control mode menu state
        state = stateMenu;
    }

    return state;
}

// Function executed on each tick in the menu state.
enum State StateHandler_Menu(const uint32_t GPIO_IF) {
    // Whether this is the first run.
    static bool firstRun = true;

    // State reported on return
    enum State state = stateMenu;
    // Whether the LEDs should blink.
    //
    // We're going to make the LEDs steady as we progress from the
    // menu, to the confirmation, and to the control mode.
    static bool led_Change0 = true,
                led_Change1 = true;


    // Blink LEDs while waiting for input
    static uint8_t led_ctr = 0;
    // Time counted before un-lighting and lighting the LEDs, in
    // 10ms ticks
    const uint8_t led_TimeUnlight    = 74,
                  led_TimeLight      = 124;
    // Switch lights on to start with
    if (firstRun) {
        GPIO_PinOutSet(gpioPortE, 2);
        GPIO_PinOutSet(gpioPortE, 3);

        firstRun = false;
    }
    // Un-light LEDs after three quarters of second
    // Re-light LEDs after a further half second
    else if (led_ctr++ == led_TimeUnlight ||
             led_ctr   == led_TimeLight    ) {

        if (led_Change0) { GPIO_PinOutToggle(gpioPortE, 2); }
        if (led_Change1) { GPIO_PinOutToggle(gpioPortE, 3); }

        if (led_ctr == led_TimeLight) {
            led_ctr = 0;
        }
    }


    // The options in the menu
    static enum {
        pwmDac,        // PWM DAC control mode
        modSupply,     // Supply modulation control mode
        fanPwm         // Fan PWM control
    } menuState = pwmDac;
    // Whether we're waiting for the user to confirm their choice
    static bool waitingConfirm = false;

    // If we're waiting for confirmation of a choice...
    if (waitingConfirm) {
        // Time, offset counters for scrolling text
        static uint8_t time = 0, offset = 0;

```

```c
        // Further instructional text
        if (time++ == 34) {
            _util_ScrollText(" PB0 CONFIRM - PB1 BACK", offset++);
            time = 0;
        }

        // If pushbutton 1 is pressed, no longer wait for
        // confirmation. We check pushbutton 1 first so that, if
        // both buttons are pressed, we default to the safe
        // option.
        if (GPIO_IF & PUSHBUTTON_1) {
            // No longer awaiting
            waitingConfirm = false;

            // We want LED0 to start blinking again
            led_Change0 = true;

            // If the LEDs have been un-lit, un-light LED0 too.
            if (led_ctr > led_TimeUnlight) {
                GPIO_PinOutClear(gpioPortE, 2);
            }

            // Reset the text position
            time = 0;
            offset = 0;

            // Display dashes to make display change clear.
            SegmentLCD_Write("-------");
        }
        // If pushbutton 0 is pressed, state transition.
        else if (GPIO_IF & PUSHBUTTON_0) {
            // Depending on the current menu selection, set the
            // state we're to enter on returning from this
            // handler.
            switch (menuState) {
                case pwmDac:    state = stateModePwmDac; break;
                case modSupply: state = stateModeSupplyModulate; break;
                case fanPwm:    state = stateModeFanPwmControl; break;
            }

            // Light both LEDs
            GPIO_PinOutSet(gpioPortE, 2);
            GPIO_PinOutSet(gpioPortE, 3);

            // No longer awaiting confirmation
            waitingConfirm = false;

            // Reset LED change settings
            led_Change0 = true;
            led_Change1 = true;

```

```
                // Reset time, offset
                time = 0;
                offset = 0;

                // Reset LED counter
                led_ctr = 0;

                // Reset first run state
                firstRun = true;

                // Clear number display
                SegmentLCD_NumberOff();

                // Clear LEDs
                GPIO_PinOutClear(gpioPortE, 2);
                GPIO_PinOutClear(gpioPortE, 3);

                // Display dashes to make display change clear.
                SegmentLCD_Write("-------");
            }
        }
        // Otherwise...
        else {
            static uint8_t time = 0, offset = 0;

            // Scroll instructional text, moving one space every 350ms
            // Space at the start to make it easier to read the first
            // character.
            if (time++ == 34) {
                _util_ScrollText(" PB0 SELECT - PB1 NEXT", offset++);
                time = 0;
            }

            // If pushbutton 0 is pressed, move to waiting for
            // confirmation.
            if (GPIO_IF & PUSHBUTTON_0) {
                // Now awaiting confirmation
                waitingConfirm = true;

                // First LED stop blinking, remain lit
                led_Change0 = false;
                GPIO_PinOutSet(gpioPortE, 2);

                // Reset time, offset
                time = 0;
                offset = 0;

                // Write dashes to LCD.
                // If the user presses the buttons at the right time,
                // it may not be clear that anything has changed (as
                // both texts start with PB0 and so may initially
```

```c
                // appear identical).
                SegmentLCD_Write("-------");
            }
            // If pushbutton 1 is pressed, move to the next option.
            else if (GPIO_IF & PUSHBUTTON_1) switch (menuState) {
                case pwmDac:    menuState = modSupply; break;
                case modSupply: menuState = fanPwm;    break;
                case fanPwm:    menuState = pwmDac;    break;
            }

            // We're a bit short on screen real estate, so we're going
            // to cheat a bit and use the numeric display. There
            // aren't many good mnemonics, but as long as they're
            // somewhat related to the control mode and not too
            // similar it should work fine.
            switch (menuState) {
                // Mnemonic: DAC as in PWM DAC
                case pwmDac: SegmentLCD_UnsignedHex(0xDAC); break;
                // Mnemonic: 5FE7 -> SFET -> Supply FET -> Supply
                // modulation
                case modSupply: SegmentLCD_UnsignedHex(0x5FE7); break;
                // Mnemonic: 57DC -> STDC -> Standard (PWM) control
                case fanPwm: SegmentLCD_UnsignedHex(0x57DC); break;
            }
        }

        return state;
}

// Function executed in the PWM DAC control mode state.
enum State StateHandler_ModePwmDac(const uint32_t GPIO_IF) {
        // Whether this is the first run.
        static bool firstRun = true;
        // Represents the duty cycle variants required by the program
        // spec.
        static enum {
            duty100, // Operate PWM DAC at 100% duty cycle
            duty60,  // Operate PWM DAC at  60% duty cycle
        } modeVariant;
        // Keeps count of elapsed 10ms ticks. Use of a uint32_t means
        // we don't have to worry about overflows, as no overflow will
        // occur for more than a year if operating continuously.
        static uint32_t ticks = 0;


        // The state reported on return
        enum State state = stateModePwmDac;


        // If this is the first run, we want to set up everything
        // needed for the PWM DAC control mode.
```

```
    if (firstRun) {
        firstRun = false;

#ifdef Config_FanEmu_PrsIncludes
        // Enable input to the pulse counter via PRS. This allows
        // us to emulate a connected fan's tachometer without
        // actually having a fan connected to the controller.
        //
        // Also configure the PRS-connected pulse generator to
        // pulse at the right rate to emulate 1200rpm.
        PCNT_PRSInputEnable(PCNT0, pcntPRSInputS0, true);
        TIMER_TopSet(TIMER3, Config_FanEmu_1200rpm);
#endif

        // The PWM DAC is connected to TIMER0 CC1 for its control
        // input, with the fan PWM input connected to TIMER1 CC2.
        // However, as we aren't using the fan PWM input in this
        // mode, we can ignore this output. A standard fan
        // operates at maximum duty cycle with no control signal.
        //
        // The CC1 output must be on PC0.

        // Configure the timer for 100% duty cycle operation. The
        // regular timer outputs a constant high signal when its
        // compare value is set above the top value.
        _util_TIMER_PrescalerSet(TIMER0, 1);
        TIMER_CompareSet(TIMER0, 1, Config_PWM_TimerCompareValue + 1);

        // Set the duty mode variable to reflect this setting
        modeVariant = duty100;

        // Route CC1 output to PC0 (which is location 4) and
        // enable output.
        TIMER0->ROUTE = TIMER_ROUTE_LOCATION_LOC4 |
                        TIMER_ROUTE_CC1PEN;

        // Start the timer and hence PWM generation
        TIMER_Enable(TIMER0, true);
    }

    // Display instructions: PB0 to go back, PB1 to go through
    // duty cycles Ticks divided by 35 so we scroll one character
    // every ~350ms
    _util_ScrollText(" PB0 BACK - PB1 DUTY", ticks / 35);

    // On the first tick and every second after that, read the
    // pulses counted by the pulse counter, convert to RPM, and
    // display the speed.
    if (ticks % 100 == 0) {
        // We use PCNT0 and, as measurements are per-second, 60
        // samples/min.
```

```
            _util_DisplayRPM(PCNT0, 60);
        }

        // Check PB1 first so that pressing both buttons defaults to
        // duty.
        if (GPIO_IF & PUSHBUTTON_1) {
            uint8_t duty = 0;

            switch (modeVariant) {
                // If we're on 100% duty cycle, switch to 60%.
                case duty100: {
                    modeVariant = duty60;
                    // Our duty cycle is out of 140, so 60% is 84/140.
                    duty = 84;

    #ifdef Config_FanEmu_PrsIncludes
                    // Set fan tachometer emulator to pulse at
                    // 1200rpm.
                    TIMER_TopSet(TIMER3, Config_FanEmu_750rpm);
    #endif
                } break;
                // If we're on 60% duty cycle, switch to 100%.
                case duty60: {
                    modeVariant = duty100;
                    duty = Config_PWM_TimerCompareValue + 1;

    #ifdef Config_FanEmu_PrsIncludes
                    // Set fan tachometer emulator to pulse at 600rpm.
                    TIMER_TopSet(TIMER3, Config_FanEmu_1200rpm);
    #endif
                } break;
            }

            TIMER_CompareSet(TIMER0, 1, duty);
        }
        // If PB0 is pressed, we want to return to the menu state.
        else if (GPIO_IF & PUSHBUTTON_0) {
            // Next state is the menu state
            state = stateMenu;

            // Reset ticks, first run
            ticks = 0;
            firstRun = true;

            // Disconnect the output channel, undo routing
            TIMER0->ROUTE = 0;

            // Disable the timer
            TIMER_Enable(TIMER0, false);

            // Clear the numbers display
```

```c
        SegmentLCD_NumberOff();
        // Display a row of dashes to make clear menu changes
        SegmentLCD_Write("-------");

#ifdef Config_FanEmu_PrsIncludes
        // Disable the PRS input to the pulse counter.
        PCNT_PRSInputEnable(PCNT0, pcntPRSInputS0, false);
#endif
    }


    // Increment ticks counter if we're remaining in this state
    if (state == stateModePwmDac)
        ticks++;

    return state;
}

// Function executed in the supply modulation control mode state.
// This function is largely the same as that for the PWM DAC.
// Refer to that function for more extensive comments.
enum State StateHandler_ModeSupplyModulate(const uint32_t GPIO_IF)
{
    static bool firstRun = true;
    // Represents the duty cycle variants required by the program
    // spec.
    static enum {
        duty100, duty80, duty60, duty40, duty15
    } modeVariant;
    // Number of 10ms ticks since entering this state.
    static uint32_t ticks = 0;


    // The state reported on return.
    enum State state = stateModeSupplyModulate;

    // If this is the first run, configure the appropriate
    // peripherals.
    if (firstRun) {
        firstRun = false;

#ifdef Config_FanEmu_PrsIncludes
        // Enable input to the pulse counter via the PRS. This
        // allows us to use the output of a timer to emulate a fan
        // tachometer at 4000rpm.
        PCNT_PRSInputEnable(PCNT2, pcntPRSInputS0, true);
        TIMER_TopSet(TIMER3, Config_FanEmu_4000rpm);
#endif

        // TIMER1 CC1 Location 4 (PD7)
        TIMER1->ROUTE = TIMER_ROUTE_LOCATION_LOC4 |
                        TIMER_ROUTE_CC1PEN;

        // Configure timer for 100% duty cycle operation at
        // 100kHz.
        _util_TIMER_PrescalerSet(TIMER1, 1);

        TIMER_CompareSet(TIMER1, 1, Config_PWM_TimerCompareValue + 1);

        // Set the duty mode variable to reflect this
        modeVariant = duty100;

        // Start the timer and PWM generation
        TIMER_Enable(TIMER1, true);
    }

    // Display instructions
    _util_ScrollText(" PB0 BACK - PB1 DUTY", ticks / 35);

    if (ticks % 100 == 0) {
        // We use PCNT0 and, as measurements are per-second, 60
        // samples/min.
        _util_DisplayRPM(PCNT2, 60);
    }

    // If PB1 is pressed, move to the next duty cycle mode.
    if (GPIO_IF & PUSHBUTTON_1) {
        uint8_t duty = 0;

        switch (modeVariant) {
            // Duty cycle 100% -> 80%
            case duty100: {
                modeVariant = duty80;
                duty = 112;

#ifdef Config_FanEmu_PrsIncludes
                TIMER_TopSet(TIMER3, Config_FanEmu_2400rpm);
#endif
            } break;

            // Duty cycle 80% -> 60%
            case duty80: {
                modeVariant = duty60;
                duty = 84;
            } break;

            // Duty cycle 60% -> 40%
            case duty60: {
                modeVariant = duty40;
                duty = 56;

#ifdef Config_FanEmu_PrsIncludes
```

```
                TIMER_TopSet(TIMER3, Config_FanEmu_4000rpm);
#endif
            } break;

            // Duty cycle 40% -> 15%
            case duty40: {
             modeVariant = duty15;
             duty = 21;
            } break;

            // Duty cycle 15% -> 100%
            case duty15: {
             modeVariant = duty100;
             duty = Config_PWM_TimerCompareValue + 1;
            } break;

        }

        TIMER_CompareSet(TIMER1, 1, duty);
    }
    // If PB0 is pressed, transition back to the control mode
    // menu.
    else if (GPIO_IF & PUSHBUTTON_0) {
        state = stateMenu;

        ticks = 0;
        firstRun = true;

        TIMER1->ROUTE = 0;

        SegmentLCD_NumberOff();
        SegmentLCD_Write("-------");


#ifdef Config_FanEmu_PrsIncludes
        PCNT_PRSInputEnable(PCNT2, pcntPRSInputS0, false);
#endif
    }

    // Increment tick counter if we're remaining in this state.
    if (state == stateModeSupplyModulate)
        ticks++;

    return state;
}

// Function executed in the standard fan PWM signal control mode
// state. This function shares a lot of similarities with the
// equivalent for the control of the PWM DAC. Refer to that
// function for more extensive comments.
enum State StateHandler_ModeFanPwmControl(const uint32_t GPIO_IF)
```

```
{
    static bool firstRun = true;
    static enum { duty100, duty70, duty30, duty20 } modeVariant;
    static uint32_t ticks = 0;


    enum State state = stateModeFanPwmControl;

    if (firstRun) {
        firstRun = false;

#ifdef Config_FanEmu_PrsIncludes
        PCNT_PRSInputEnable(PCNT2, pcntPRSInputS0, true);
        TIMER_TopSet(TIMER3, Config_FanEmu_600rpm);
#endif

        // Using fan PWM control is slightly different, as we need
        // to use both the supply-modulating MOSFET and the
        // control signal MOSFET. As the supply MOSFET only
        // controls the connection to ground, we can just
        // provide a constant output on a GPIO to control it.
        //
        // PWM control is on LETIMER0 OUT1 Location 1 (PB11).
        // Supply modulation signal is on PD7.

        // The LETIMER is preconfigured for 25kHz and isn't used
        // for any other function, so we only need to reset the
        // duty cycle.
        LETIMER_CompareSet(LETIMER0, 1, Config_PWM_LETimerCompareValue);

        // Set the mode variable accordingly
        modeVariant = duty100;

        // Route to location 1 and connect output 1
        LETIMER0->ROUTE = LETIMER_ROUTE_LOCATION_LOC1 |
                          LETIMER_ROUTE_OUT1PEN;

        // GPIO is already configured, so we just need to set it
        // high.
        GPIO_PinOutSet(gpioPortD, 7);

        // Start the timer and enable PWM generation
        LETIMER_Enable(LETIMER0, true);
    }

    // Instructions
    _util_ScrollText(" PB0 BACK - PB1 DUTY", ticks / 35);

    // RPM display
    if (ticks % 100 == 0) {
        // We use PCNT0 and, as measurements are per-second, 60
        // samples/min.
        _util_DisplayRPM(PCNT2, 60);
    }

    // If PB1 is pressed, move to the next duty cycle mode.
    if (GPIO_IF & PUSHBUTTON_1) {
        uint8_t duty = 0;

        switch (modeVariant) {
            // Duty cycle 100% -> 70%
            case duty100: {
                modeVariant = duty70;
                duty = 98; // 98/140 == 70%

#ifdef Config_FanEmu_PrsIncludes
                TIMER_TopSet(TIMER3, Config_FanEmu_420rpm);
#endif
            } break;

            // Duty cycle 70% -> 30%
            case duty70: {
                modeVariant = duty30;
                duty = 42; // 42/140 == 30%

                // PRS emulation in this additional duty cycle
                // mode isn't possible, as it would require
                // changing more than just the emulation timer's
                // TOP value to achieve low enough frequency.
            } break;

            // Duty cycle 30% -> 20%
            case duty30: {
                modeVariant = duty20;
                duty = 28; // 28/140 == 20%

                // PRS emulation in this additional duty cycle
                // mode isn't possible, as it would require
                // changing more than just the emulation timer's
                // TOP value to achieve low enough frequency.
            } break;

            // Duty cycle 20% -> 100%
            case duty20: {
                modeVariant = duty100;
                duty = Config_PWM_LETimerCompareValue;

#ifdef Config_FanEmu_PrsIncludes
                TIMER_TopSet(TIMER3, Config_FanEmu_600rpm);
#endif
            } break;
        }

        LETIMER_CompareSet(LETIMER0, 1, duty);
    }
    // If PB0 is pressed, transition back to the control mode
    // menu.
    else if (GPIO_IF & PUSHBUTTON_0) {
        state = stateMenu;

        ticks = 0;
        firstRun = true;

        LETIMER0->ROUTE = 0;

        SegmentLCD_NumberOff();
        SegmentLCD_Write("-------");

        // Disable the supply modulation MOSFET gate signal GPIO
        GPIO_PinOutClear(gpioPortD, 7);

#ifdef Config_FanEmu_PrsIncludes
        PCNT_PRSInputEnable(PCNT2, pcntPRSInputS0, false);
#endif
    }

    if (state == stateModeFanPwmControl)
        ticks++;

    return state;
}

// Utility function for scrolling text. See declaration for more
// info.
void _util_ScrollText(const char* text, uint32_t offset) {
      // The length of the string
      const size_t len = strlen(text);

      SegmentLCD_Write( text + (offset % len) );
}

// Utility function for displaying PCNT counter on the numeric
// LCD. See the declaration for parameter info.
void _util_DisplayRPM(PCNT_TypeDef* pcnt, const uint8_t samplesPerMin) {
    // The cast to int shouldn't affect the result, as the pulse
    // counter is only able to count up to 255 and storing 255 in
    // an [int16_t] won't get close to altering the sign bit.
    SegmentLCD_Number(
        (int16_t)PCNT_CounterGet(pcnt) * (samplesPerMin / 2)
    );

    PCNT_CounterReset(pcnt);
}
```

```c

// Helper function for adjusting the prescaler for a TIMER.
//
// Parameters:
//  timer       = The timer the prescaler of which to set.
//  divisor     = The amount by which the timer clock is to be
//                divided. Valid values are powers of 2 from 1 to
//                1024 (i.e. 2**0 to 2**10).
//
// Returns:
//  False if the divisor is greater than 1024.
//  True if the setting is applied.
//
// Remarks:
//  If the provided divisor is not a power of two, the value
//  represented by the most significant set bit will be taken as
//  the specified value.
//
// Declared in states.h.
bool _util_TIMER_PrescalerSet(TIMER_TypeDef* timer,
                              const uint8_t divisor) {
    // Indicate failure if the divisor is greater than 1024.
    if (divisor > 1024) {
        return false;
    }

    // The prescaler bits are stored in TIMERx_CTRL[27:24],
    // requiring the bits  be shifted up 24 positions.
    const uint32_t prescVal = CMU_DivToLog2(divisor) << 24;

    // Clear the prescaler value from the register and set the new
    // value.
    timer->CTRL = (timer->CTRL & 0xF0FFFFFFU) | prescVal;

    return true;
}
```

C3 USB Prototype — Firmware

The following is the source code for the USB prototype firmware, written in C. The firmware emulates a fan controller which implements the Appendix D protocol and which controls a single fan. Discussion on the firmware is in Appendix G1.3.

As noted in that discussion, certain corrections were made to code provided by the vendor. The corrected code is not included here, but the changes made are described in that discussion.

main.c

The firmware entry point and general microcontroller configuration.

```c
/* Author: Liam McSherry
 * Date:   22nd February 2018
 * Notes:  Entry point and configuration, etc.
*/

// Standard C headers
#include <stdbool.h>
#include <stdint.h>
// Device-specific headers for the EFM32WG990.
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_timer.h"
#include "segmentlcd.h"
// Corrected/etc USB middleware headers
#include "emusb-modheaders/em_usb.h"
// Headers from the development kit board support package.
#include "bsp.h"
// Firmware-specific headers
#include "callbacks.h"

// Local function prototypes
void configure_Clocks(void);
void configure_USB(void);

int main(void)
{
    // Initialisation, erratum fixes, calibration, etc.
    CHIP_Init();

    configure_Clocks();
    configure_USB();

    /* Infinite loop */
    while (1);
}
```

```c
// Configures clocks for the chip.
void configure_Clocks(void) {

    // The USB controller requires HFCORE to operate at 48MHz.
    // The development kit provides a 48MHz crystal, so we drive
    // HFCORE from that crystal.
    //
    // Enable access to LE peripherals when >24MHz. Other
    // settings, such as the buffer current setting and mode
    // setting, are already handled by the default register
    // values.
    CMU->CTRL |= CMU_CTRL_HFLE;
    // Set HFCORE divider for operation above 24MHz.
    CMU->HFCORECLKDIV |= CMU_HFCORECLKDIV_HFCORECLKLEDIV_DIV4;
    // Enable HFXO
    CMU_OscillatorEnable(cmuOsc_HFXO, true, true);

    // Spinwait until HFXO is ready
    while ( !(CMU->STATUS & CMU_STATUS_HFXORDY) ) ;

    // Set HFXO as the HFCORE driver and HFCLK as the
    // HFCORECLK(USBC) source.
    CMU->CMD = CMU_CMD_HFCLKSEL_HFXO |
               CMU_CMD_USBCCLKSEL_HFCLKNODIV;


    // Configure HFPER to operate at 48MHz/16 == 3MHz and
    // enable.
    CMU_ClockDivSet(cmuClock_HFPER, cmuClkDiv_16);
    CMU_ClockEnable(cmuClock_HFPER, true);

    // Enable clock for timer 0.
    CMU_ClockEnable(cmuClock_TIMER0, true);

    // Enable clock for GPIO.
    CMU_ClockEnable(cmuClock_GPIO, true);
}

// Configure the USB controller
void configure_USB(void) {

    // The initialisation function for the USB controller uses a
    // number of pointers to structs. These must therefore be
    // static variables.
    //
    // Documentation for functions in [em_usbd.c] commonly states
    // that variables/buffers must be 4-byte aligned, so we're
    // going to align everything here to be safe.

    static const USB_DeviceDescriptor_TypeDef
        usb_device __attribute__ (( aligned(4) )) = {
            // This is a device descriptor, so the length is
            // constant and provided by the driver.
            .bLength            = USB_DEVICE_DESCSIZE,
            .bDescriptorType    = USB_DEVICE_DESCRIPTOR,
            // We're supporting USB 2.1 so we can use Binary
            // Device Object Store (BOS) descriptors, which are
            // required to autoload WinUSB.
            .bcdUSB             = 0x0210,
            // Device class information is defined on a per-
            // -interface basis, and not at device level.
            .bDeviceClass       = 0x00,
            .bDeviceSubClass    = 0x00,
            .bDeviceProtocol    = 0x00,
            // We're going to accept packets of 64 bytes on
            // the control endpoint 0.
            .bMaxPacketSize0    = 64,
            // VID and PID for the device
            .idVendor           = USB_VID,
            .idProduct          = USB_PID,
            // Version number for the device, doesn't need to
            // be meaningful so we'll state v0.1.
            .bcdDevice          = 0x0001,
            // Indices of string descriptors for strings that
            // contain the manufacturer and product names and
            // the serial number.
            .iManufacturer      = 1,
            .iProduct           = 2,
            .iSerialNumber      = 3,
            // The number of configurations for the device
            .bNumConfigurations = 1
    };

    // Language ID string descriptor
    static const struct __attribute__ (( aligned(4) )) {
        uint8_t bLength;
        uint8_t bDescriptorType;
        uint8_t wLangID[2];
    } iLangID = {
        4,
        USB_STRING_DESCRIPTOR,
        // English (United Kingdom) language ID.
        // Actual value is 0x0809, but multibyte fields must be
        // stored in little-endian order per USB section 8.1.
        { 0x09, 0x08 }
    };

    // iManufacturer string descriptor
    static const struct __attribute__ (( aligned(4) )) {
        uint8_t  bLength;
        uint8_t  bDescriptorType;
        char16_t bString[13];
    } iManufacturer = {
            28,
            USB_STRING_DESCRIPTOR,
            {
                'L', 'i', 'a', 'm', ' ',
                'M', 'c', 'S', 'h', 'e', 'r', 'r', 'y'
            }
    };

    // iProduct string descriptor
    static const struct __attribute__ (( aligned(4) )) {
        uint8_t  bLength;
        uint8_t  bDescriptorType;
        char16_t bString[23];
    } iProduct = {
            48,
            USB_STRING_DESCRIPTOR,
            {
                'G', 'U', '2', ' ', 'U', 'S', 'B', ' ',
                'F', 'i', 'r', 'm', 'w', 'a', 'r', 'e', ' ',
                'D', 'e', 'v', 'i', 'c', 'e'
            }
    };

    // iSerialNumber string descriptor
    static const struct __attribute__ (( aligned(4) )) {
        uint8_t  bLength;
        uint8_t  bDescriptor;
        char16_t bString[1];
    } iSerialNumber = {
            4,
            USB_STRING_DESCRIPTOR,
            { 'A' }
    };

    // Identifier string descriptor for the fan we're pretending
    // is connected to the controller
    static const struct __attribute__ (( aligned(4) )) {
        uint8_t  bLength;
        uint8_t  bDescriptor;
        char16_t bString[13];
    } iFanCtrlIdentifier = {
        28,
        USB_STRING_DESCRIPTOR,
        {
            'F', 'a', 'n', ' ', '#', '1', ' ',
            '(', 'E', 'm', 'u', '.', ')'
        }
    };

    // Array of string descriptors
    static const void* const usb_strings[5] = {
            &iLangID,           // 0
            &iManufacturer,     // 1
            &iProduct,          // 2
            &iSerialNumber,     // 3
            &iFanCtrlIdentifier // 4
    };

    static const uint8_t usb_config[]
        __attribute__ (( aligned(4) )) = {
        /***** Standard Configuration Descriptor *****/
        /* bLength */               USB_CONFIG_DESCSIZE,
        /* bDescriptorType */       USB_CONFIG_DESCRIPTOR,
        // The length in bytes of all descriptors returned
        // with the standard configuration descriptor.
        /* wTotalLength */          26, 0,
        /* bNumInterfaces */        1,
        /* bConfigurationValue */   1,
        // Index of a string descriptor describing this
        // configuration, 0 means no descriptor.
        /* iConfiguration */        0,
        /* bmAttributes */          CONFIG_DESC_BM_RESERVED_D7 |
                                    CONFIG_DESC_BM_SELFPOWERED,
        /* bmMaxPower */            CONFIG_DESC_MAXPOWER_mA(100),

        /***** Standard Interface Descriptor *****/
        /* bLength */               USB_INTERFACE_DESCSIZE,
        /* bDescriptorType */       USB_INTERFACE_DESCRIPTOR,
        /* bInterfaceNumber */      0,
        /* bAlternateSetting */     0,
        /* bNumEndpoints */         NUM_EP_USED,
        // The interface uses a vendor-defined device class
        /* bInterfaceClass */       0xFF,
        // The subclass and protocol don't need to have any
        // meaning since this isn't a "real" device.
        /* bInterfaceSubclass */    0x00,
        /* bInterfaceProtocol */    0x00,
        /* iInterface */            0,

        /***** Standard Endpoint Descriptor *****/
        // We're only using endpoint zero, so no endpoint
        // descriptor is to be provided (per USB s. 9.6.6).

        /**********/
        // Microsoft-specific descriptors are retrieved by
        // using a Microsoft-defined request, and so are not
        // included with these descriptors.

        /***** Appendix D Protocol Descriptor *****/
        // This is the fan controller configuration descriptor
```

```c
        // defined by Appendix D.
        /* bLength */               8,
        /* bDescriptorType */       0x20,
        /* bmAttributes */          0x00,       // 1x fan support
        /* bcdVersion */            0x11, 0x00, // v0.11 protocol
        /* Reserved */              0, 0, 0,
    };

    // We don't want to care here about the specifics of the
    // callbacks, we just want something we can initialise the
    // middleware with. Note, this is already a pointer, so we
    // don't need to take its address in the struct assignment.
    const USBD_Callbacks_TypeDef* usb_cbacks = getUsbCallbacks();

    static const uint8_t bufferingMultiplier = 3;

    const USBD_Init_TypeDef usbd_init = {
            .deviceDescriptor       = &usb_device,
            .configDescriptor       = usb_config,
            .stringDescriptors      = usb_strings,
            .numberOfStrings        = 4,
            .callbacks              = usb_cbacks,
            .bufferingMultiplier    = &bufferingMultiplier,
            .reserved               = 0
    };


    USBD_Init(&usbd_init);

    USBD_Connect();
}
```

callbacks.c

The code containing the callbacks used by the vendor-provided USB device stack which handle USB-related events (such as receiving a request).

```c
/* Author: Liam McSherry
 * Date:   3rd March 2018
 * Notes:  Provides the callbacks used by the USB driver and
 *         related logic for the firmware.
*/


// Standard C headers
#include <stdlib.h>
#include <stdbool.h>
// Corrected/etc USB middleware headers
#include "emusb-modheaders/em_usb.h"
// Firmware-specific headers
#include "callbacks.h"

```

```c
// Local prototypes
int MiddlewareCallback_SetupCmd(const USB_Setup_TypeDef* setup);
int ReqHandler_GetDescriptor_BOS(const USB_Setup_TypeDef* setup);
int ReqHandler_MsDescriptorSet(const USB_Setup_TypeDef* setup);
int ReqHandler_GetFanID(const USB_Setup_TypeDef* setup);
int ReqHandler_SetFanMode(const USB_Setup_TypeDef* setup);
int ReqHandler_SetFanMode_ProcessData(
    USB_Status_TypeDef, uint32_t, uint32_t);
int ReqHandler_GetFanMode(const USB_Setup_TypeDef* setup);


// Returns a pointer to the callbacks struct.
const USBD_Callbacks_TypeDef* getUsbCallbacks(void) {
    // Used by the USB driver to call back to application code
    // to enable that code to handle USB events.
    static const USBD_Callbacks_TypeDef
        usb_callbacks __attribute__ (( aligned(4) )) = {
            // Callback for handling a reset by the host.
            .usbReset       = NULL,
            // Callback for handling a change in device state.
            .usbStateChange = NULL,
            // Callback for handling a SETUP packet from the USB
            // host, can be used to override handling for standard
            // requests as well as class-/vendor-defined requests.
            .setupCmd       = &MiddlewareCallback_SetupCmd,
            // Callback to enable the driver to determine whether
            // the device is self-powered at any given time.
            .isSelfPowered  = NULL,
            // Callback for the detection of a Start-of-Frame
            // packet, which are regularly issued at 1000Hz (full
            // speed) or 8kHz (high speed) (per USB s. 8.4.3).
            .sofInt         = NULL
    };

    return &usb_callbacks;
}


// Callback for handling SETUP packets.
//
// Parameters:
//  setup   = The contents of the SETUP packet, from the driver.
//            See USB section 9.3 (table 9-2).
//
// Returns:
//  A status code that is a [USB_Status_TypeDef] request status
//  code value.
int MiddlewareCallback_SetupCmd(const USB_Setup_TypeDef* setup) {
    // If we don't handle a request, we can indicate that we
    // didn't and the middleware will attempt to handle it for us.
    int status  = USB_STATUS_REQ_UNHANDLED;

    // To indicate we want WinUSB loaded on the host, we need to
    // respond to a request for the USB BOS descriptor and include
    // with that descriptor the platform-specific descriptors used
    // by the Microsoft OS Descriptors 2.0 specification.
    if (setup->Type == USB_SETUP_TYPE_STANDARD) {

        // Handle GET_DESCRIPTOR requests for the BOS descriptor
        if (setup->bRequest == GET_DESCRIPTOR &&
            setup->wValue   == (USB_BOS_DESCRIPTOR << 8)) {
            status = ReqHandler_GetDescriptor_BOS(setup);
        }

    }
    // The operating system will send a vendor-specific request,
    // using the values we reported in the BOS, to retrieve the
    // operating system-specific descriptors.
    else if (setup->Type == USB_SETUP_TYPE_VENDOR) {
        switch (setup->bRequest) {

            // A request of type [bMS_VendorCode] is used by the
            // host to retrieve MS OS 2.0 feature descriptors.
            case bMS_VendorCode: {
                status = ReqHandler_MsDescriptorSet(setup);
            } break;

        }
    }
    // The requests defined in the fan controller device class
    // specification are class-specific.
    else if (setup->Type == USB_SETUP_TYPE_CLASS) {
        switch (setup->bRequest) {

            case FANCTRL_FANID_REQ: {
                status = ReqHandler_GetFanID(setup);
            } break;

            case FANCTRL_MDSET_REQ: {
                status = ReqHandler_SetFanMode(setup);
            } break;

            case FANCTRL_MDGET_REQ: {
                status = ReqHandler_GetFanMode(setup);
            } break;

        }
    }

    return status;
}

```

```
// Implements USB BOS descriptor handling
int ReqHandler_GetDescriptor_BOS(const USB_Setup_TypeDef* setup) {
    // All Binary Device Object Store (BOS) descriptors. These
    // are used to indicate that we support MS OS Descriptors 2.0
    // and so to tell the host that it can query those.
    static const uint8_t
        bosDescriptors[50] __attribute__ (( aligned(4) )) = {

        /***** BOS Descriptor *****/
        /* bLength */                   USB_BOS_DESCSIZE,
        /* bDescriptorType */           USB_BOS_DESCRIPTOR,
        /* wTotalLength */              50, 0x00,
        /* bDeviceNumCaps */            3,

        /***** USB 2.0 Extension Descriptor *****/
        // See USB 3.1 section 9.6.2.1
        /* bLength */                   7,
        /* bDescriptorType */           USB_BOS_DEVCAP_DESC,
        /* bDevCapabilityType */        BOS_USB2_EXTENSION,
        /* bmAttributes */              0, 0, 0, 0,

        /***** SuperSpeed USB Descriptor *****/
        // See USB 3.1 section 9.6.2.2
        /* bLength */                   10,
        /* bDescriptorType */           USB_BOS_DEVCAP_DESC,
        /* bDevCapabilityType */        BOS_SUPERSPEED_USB,
        /* bmAttributes */              0,
        /* wSpeedsSupported */          0x03, 0x00,
        /* bFunctionalitySupport */     0x00, // Fine at low speed
        // These values are for Link Power Management, which we
        // don't appear to need to support, so we zero them.
        /* bU1DevExitLat */             0,
        /* wU2DevExitLat */             0, 0,

        /***** Device Capability Descriptor 1 *****/
        // These fields are from the USB standard
        // Indicates Microsoft OS 2.0 Descriptor support
        /* bLength */                   28,
        /* bDescriptorType */           USB_BOS_DEVCAP_DESC,
        /* bDevCapabilityType */        BOS_CAP_PLATFORM,
        // The following fields are from the Microsoft standard
        /* bReserved */                 0,
        // UUID {D8DD60DF-4589-4CC7-9CD2-659D9E648A9F} indicates
        // that Microsoft OS 2.0 Descriptors is supported. As
        // above, multibyte fields are stored little-endian
        /* PlatformCapabilityUUID */    0xDF, 0x60, 0xDD, 0xD8,
                                        0x89, 0x45, 0xC7, 0x4C,
                                        0x9C, 0xD2, 0x65, 0x9D,
                                        0x9E, 0x64, 0x8A, 0x9F,
        // The descriptor set information structure contained in
        // this section indicates to which version of Windows the
        // descriptor set applies, provides the total set length,
        // the [bMS_VendorCode] value used to retrieve feature
        // descriptors, and other data.
        /* CapabilityData */
        // Windows version is Windows Blue or later
        /* dwWindowsVersion */          NTDDI_WINBLUE_LE,
        /* DescriptorSetTotalLength */  0x0A+132+0x14, 0,
        /* bMS_VendorCode */            bMS_VendorCode,
        /* bAltEnumCode */              0

    };

    // Ensure we don't transfer more data than we're meant to by
    // clipping the value of [wLength] if it's too great
    int byteCount = setup->wLength > sizeof(bosDescriptors)
                        ? sizeof(bosDescriptors)
                        : setup->wLength;

    // Transfer the BOS descriptors
    return USBD_Write(0, &bosDescriptors, byteCount, NULL);
}

// Handles requests for the Microsoft OS 2.0 descriptor set
int ReqHandler_MsDescriptorSet(const USB_Setup_TypeDef* setup) {
    static const uint8_t msosDescSet[0x0A+132+0x14]
        __attribute__ (( aligned(4) )) = {

        /***** Descriptor Set Header *****/
        /* wLength */           MS_SETHEADER_DESCSZ,
        /* wDescriptorType */   MS_SETHEADER_DESC,
        /* dwWindowsVersion */  NTDDI_WINBLUE_LE,
        /* wTotalLength */      0x0A+132+0x14, 0,


        /***** Registry Property Descriptor *****/
        // This descriptor is used to set a device interface GUID
        // in the registry, so we can access the device.
        /* wLength */               132, 0,
        /* wDescriptorType */       MS_REGISTRY_DESC,
        /* wPropertyDataType */     REGISTRY_REG_ML_SZ,
        /* wPropertyNameLength */   42, 0,
        // UTF-16-encoded "DeviceInterfaceGUIDs" (+NUL-term.)
        /* wPropertyName */         0x44, 0x00, 0x65, 0x00,
                                    0x76, 0x00, 0x69, 0x00,
                                    0x63, 0x00, 0x65, 0x00,
                                    0x49, 0x00, 0x6E, 0x00,
                                    0x74, 0x00, 0x65, 0x00,
                                    0x72, 0x00, 0x66, 0x00,
                                    0x61, 0x00, 0x63, 0x00,
                                    0x65, 0x00, 0x47, 0x00,
                                    0x55, 0x00, 0x49, 0x00,
                                   0x44, 0x00, 0x73, 0x00,
                                   0x00, 0x00,
        /* wPropertyDataLength */   80, 0,
        // The GUID 0A56B842-14F1-11E8-BA16-00805FC181FE encoded
        // as UTF-16, with enclosing braces {}.
        /* PropertyData */         0x7B, 0x00, 0x30, 0x00,
                                   0x41, 0x00, 0x35, 0x00,
                                   0x36, 0x00, 0x42, 0x00,
                                   0x38, 0x00, 0x34, 0x00,
                                   0x32, 0x00, 0x2D, 0x00,
                                   0x31, 0x00, 0x34, 0x00,
                                   0x46, 0x00, 0x31, 0x00,
                                   0x2D, 0x00, 0x31, 0x00,
                                   0x31, 0x00, 0x45, 0x00,
                                   0x38, 0x00, 0x2D, 0x00,
                                   0x42, 0x00, 0x41, 0x00,
                                   0x31, 0x00, 0x36, 0x00,
                                   0x2D, 0x00, 0x30, 0x00,
                                   0x30, 0x00, 0x38, 0x00,
                                   0x30, 0x00, 0x35, 0x00,
                                   0x46, 0x00, 0x43, 0x00,
                                   0x31, 0x00, 0x38, 0x00,
                                   0x31, 0x00, 0x46, 0x00,
                                   0x45, 0x00, 0x7D, 0x00,
                                   0x00, 0x00, 0x00, 0x00,

        /***** Compatible ID Header *****/
        // This descriptor is used to indicate that we want the
        // operating system to load WinUSB for us.
        /* wLength */           MS_COMPATID_DESCSZ,
        /* wDescriptorType */   MS_COMPATID_DESC,
        // The text "WINUSB" in ASCII
        /* CompatibleID */      0x57, 0x49, 0x4E, 0x55,
                                0x53, 0x42,
        // No value
        /* SubcompatibleID */   0x00, 0x00, 0x00, 0x00,
                                0x00, 0x00, 0x00, 0x00,
    };

    if (setup->wIndex == 0x07) {
        int byteCount = setup->wLength > sizeof(msosDescSet)
                        ? sizeof(msosDescSet)
                        : setup->wLength;

        return USBD_Write(0, &msosDescSet, byteCount, NULL);
    }

    return USB_STATUS_REQ_UNHANDLED;
}
```

```c
// Buffer for storing received mode data from the host.
static uint8_t modeBuf[32] __attribute__ (( aligned(4) ));
// The length of [modeBuf]. Zero if we haven't been configured.
static size_t  modeBufSize = 0;

// Handles requests for fan identification/status information,
// and responds with dummy values.
int ReqHandler_GetFanID(const USB_Setup_TypeDef* setup) {
    static uint8_t fanId[8] __attribute__ (( aligned(4) )) = {
        /* bmAttributes */  0xD3, 0x07, 0x7D, // Voltage Control
                                              // Speed Control
                                              // Start @ 7.5V
                                              // 30% min speed
                                              // 4000rpm max
        /* wCurrent */      0x8C, 0x00,       // 140mA
        /* iIdentifier */   4,
        /* Reserved */      0,
    };

    // Pointer to wCurrent so we can more easily fiddle with it
    // between requests.
    static uint16_t * const wCurrent = (uint16_t*)(fanId + 3);
    // Counter so we can use a repeating pattern with requests.
    static uint16_t counter = 0;

    // If the request is for a fan other than the one fan that
    // we support, return a request error.
    if (setup->wValue != 0)
        return USB_STATUS_REQ_ERR;

    // If we haven't received a SET_FAN_MODE request, rotate
    // through four constant predefined currents.
    if (modeBufSize == 0) switch (counter++ % 4) {
        case 0: *wCurrent = 0x008D; break; // 141mA
        case 1: *wCurrent = 0x008E; break; // 142mA
        case 2: *wCurrent = 0x008B; break; // 139mA
        case 3: *wCurrent = 0x008C; break; // 140mA
    }
    // If we've previously received a SET_FAN_MODE request, then
    // report the configured temperatures as our currents.
    else {
        // We need to keep track of our index across requests.
        static uint8_t callNo = 0;

        // If we've reached the end of the array, reset the index.
        if (callNo >= (modeBufSize / 2))
            callNo = 0;

        // Each entry is two bytes, so to iterate through every
        // second byte starting at zero (as the first byte of a
        // two-byte entry contains the temperature) we need to
        // multiply the index by two.
        //
        // The first bit of an entry is a last-in-sequence
        // indicator, so we need to shift that away.
        *wCurrent = modeBuf[callNo++ * 2] >> 1;
    }

    // Return the buffer
    return USBD_Write(0, &fanId, 8, NULL);
}

// Handles requests to set fan configuration mode.
int ReqHandler_SetFanMode(const USB_Setup_TypeDef* setup) {
    // The format of the contents of [wValue].
    struct wValue_TypeDef {
        uint8_t fan     : 4; // Zero-based fan identifier
        uint8_t mode    : 2; // Mode specifier
    } *wValue;


    // A basic sanity check--if the data we're provided with is
    // going to be more than 32 bytes long, we'll reject it. In
    // testing, anything more than this probably indicates that
    // something has gone wrong.
    //
    // If this were a real application, the specification would
    // probably include a limit on the number of points that a
    // user could specify.
    if (setup->wLength > sizeof(modeBuf))
        return USB_STATUS_REQ_ERR;

    // Sugar for accessing the fields we want.
    wValue = (struct wValue_TypeDef*) &(setup->wValue);

    // As with the software, we're only implementing the code
    // for handling voltage mode data. Accordingly, we have to
    // fail if speed mode data or anything else is provided.
    if (wValue->mode != FANCTRL_MODE_VOLTS)
        return USB_STATUS_REQ_ERR;

    // We've only got the one fan, so we can hardcode the check
    // of the fan identifier, but in real-world code this would
    // likely be a check against the number of fans supported.
    if (wValue->fan != 0)
        return USB_STATUS_REQ_ERR;

    // In real code, this would be a switch and a set of calls to
    // some split-out verification functions. However, since we
    // only handle voltage mode data, it's included here.
    //
    // If the length of the data is odd, we can be sure that the
    // data is wrong, as each entry must be two bytes long. The
    // same applies if there is no data.
    if ( setup->wLength == 0 || (setup->wLength % 2) == 1 )
        return USB_STATUS_REQ_ERR;

    // We now need to iterate through the data, verifying each
    // individual entry. The first step in this is reading the
    // data into the buffer.
    USBD_Read(
        0, &modeBuf, setup->wLength,
        ReqHandler_SetFanMode_ProcessData
        );

    return USB_STATUS_OK;
}

// Callback for handling data read by the [ReqHandler_SetFanMode]
// function. This is effectively an extension of that method.
int ReqHandler_SetFanMode_ProcessData(
    USB_Status_TypeDef status,
    uint32_t           xferred,
    uint32_t           remaining
    ) {

    // The format of each voltage control mode entry.
    struct vcmEntry_TypeDef {
        bool    end     : 1; // Whether this is the last entry
        uint8_t temp    : 7; // The temperature, degrees C.
        uint8_t voltage    ; // The controlled voltage
    } *vcmEntry;
    if (status != USB_STATUS_OK)
        return status;

    if (remaining > 0)
        return USB_STATUS_REQ_ERR;

    // We then want to view the buffer as a set of entries, and
    // iterate through the set (verifying as we go).
    const int numEntries = xferred / 2;
    vcmEntry = (struct vcmEntry_TypeDef*) &modeBuf;

    for (int i = 0; i < numEntries; i++) {
        // It is impossible for temperature and voltage data to
        // be invalid, as the defined range of valid values and
        // the possible range of values are the same. As such,
        // all we need to verify is that the END bit is correct.
        //
        // This is done by checking that the END bit is only set
        // if this is the last index (which, given arrays are
        // zero-indexed, is one less than the length).
        if (i != (numEntries - 1) && vcmEntry[i].end)
```

```c
                return USB_STATUS_REQ_ERR;
            // And also by checking that the last entry does have the
            // END bit set.
            else if (i == (numEntries - 1) && !vcmEntry[i].end)
                return USB_STATUS_REQ_ERR;
        }

        modeBufSize = xferred;

        // If we ended up here, then the request succeeded.
        return USB_STATUS_OK;
    }

    // Handles requests for the mode data
    int ReqHandler_GetFanMode(const USB_Setup_TypeDef* setup) {
        // The format of the contents of [wValue].
        struct wValue_TypeDef {
            uint8_t fan     : 4; // Zero-based fan identifier
        } *wValue;

        // If we haven't been configured, there isn't anything to
        // return to the host. This wouldn't happen with a real-world
        // application, as there would be a default configuration.
        if (modeBufSize == 0)
            return USB_STATUS_REQ_ERR;

        wValue = (struct wValue_TypeDef*) &(setup->wValue);

        // Again, we're hardcoding a value here for ease in testing.
        if (wValue->fan != 0)
            return USB_STATUS_REQ_ERR;

        // Clip the requested size if it's longer than our data
        int byteCount = setup->wLength > modeBufSize
                            ? modeBufSize
                            : setup->wLength;

        return USBD_Write(0, &modeBuf, byteCount, NULL);
    }
```

usbconfig.h

A header containing USB-related configuration for both the vendor-provided middleware and the firmware.

```c
/* Author: Liam McSherry
 * Date:   1st March 2018
 * Notes:  USB configuration information for the driver. Despite
 *         it being a relatively important piece of the puzzle,
 *         documentation for it seems relatively scarce.
*/
#ifndef SRC_USBCONFIG_H_
```

```c
#define SRC_USBCONFIG_H_


/***** DRIVER CONFIGURATION *****/

// Driver in USB device mode
#define USB_DEVICE

// Driver emits debug information on serial
#define DEBUG_USB_API

// Driver uses TIMER0 for its timer
#define USB_TIMER           USB_TIMER0

#define NUM_APP_TIMERS      1


/***** DEVICE CONFIGURATION *****/

// No USB endpoints additional to the control endpoint used
#define NUM_EP_USED         0


/***** FIRMWARE CONFIGURATION *****/

// Vendor and Product IDs for the USB device.
//
// 1209:0001 is a pair for private use allocated by pid.codes,
// which we'll be using to avoid obtaining an allocation.
#define USB_VID             0x1209
#define USB_PID             0x0001

// Request codes for fan controller class-specific requests
#define FANCTRL_FANID_REQ   0   // GET_FAN_ID
#define FANCTRL_MDSET_REQ   1   // SET_FAN_MODE
#define FANCTRL_MDGET_REQ   2   // GET_FAN_MODE

// Mode data mode specifiers
#define FANCTRL_MODE_VOLTS  0x00
#define FANCTRL_MODE_SPEED  0x01


// Descriptor type value for the Binary Device Object Store (BOS)
// descriptor described in the USB 3.1 specification.
#define USB_BOS_DESCRIPTOR  15
// The length of the BOS descriptor, in bytes.
#define USB_BOS_DESCSIZE    5
// USB BOS Device Capability descriptor
#define USB_BOS_DEVCAP_DESC 16

```

```
// Binary Device Object Store (BOS) capability type codes
#define BOS_USB2_EXTENSION  0x02    // USB 2.0 EXTENSION
#define BOS_SUPERSPEED_USB  0x03    // SUPERSPEED_USB
#define BOS_CAP_PLATFORM    0x05    // PLATFORM

// BOS SUPERSPEED_USB descriptor definitions
#define BOS_SSUSB_SPEED_LS  1 << 0  // Low Speed
#define BOS_SSUSB_SPEED_FS  1 << 1  // Full Speed

// The value reported to the host for use in retrieving the MS
// OS 2.0 descriptor set.
#define bMS_VendorCode      0xA5

// Microsoft OS 2.0 descriptor types
// Descriptor set header
#define MS_SETHEADER_DESC   0x00, 0x00
#define MS_SETHEADER_DESCSZ 0x0A, 0x00
// Compatible ID descriptor
#define MS_COMPATID_DESC    0x03, 0x00
#define MS_COMPATID_DESCSZ  0x14, 0x00
// Registry property descriptor
#define MS_REGISTRY_DESC    0x04, 0x00
#define REGISTRY_REG_SZ     0x01, 0x00 // REG_SZ type
#define REGISTRY_REG_ML_SZ  0x07, 0x00 // REG_MULTI_SZ


// Little-endian formatted NTDDI Windows 10 specifier.
#define NTDDI_WIN10_LE      0x00, 0x00, 0x00, 0x0A
#define NTDDI_WINBLUE_LE    0x00, 0x00, 0x03, 0x06

#endif /* SRC_USBCONFIG_H_ */
```

C4 USB Prototype — Software

The following is the source code for the USB prototype software, written in C# and targeting the .NET Framework. The software provides a user interface which enables control of the fan controller represented by the USB prototype firmware. Discussion on the software is contained in Appendix G1.5.

Main.cs

The software entry point, responsible for processing user input and instructing the fan controller through the exposed interface.

```
/* Author:  Liam McSherry
 * Date:    3rd March 2018
 * Notes:   Entry point to the USB prototype software, includes
 *          basic setup and related.
 */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Timers;

using Windows.Devices.Enumeration;
using Windows.Devices.Usb;

namespace McSherry.Edu.HND.UsbProto
{
    class Program
    {
        // The Vendor and Product IDs of the device.
        private const uint VID = 0x1209, PID = 0x0001;


        // Timer for requesting fan status updates.
        private static Timer _updateTimer;
        private static FanController _fanCtrl;

        static void Main(string[] args)
        {
            while (true)
            {
                MainAsync(args).Wait();
            }
        }

        static async Task MainAsync(string[] args)
        {
            var aqs = UsbDevice.GetDeviceSelector(VID, PID);
            var devs = await DeviceInformation.FindAllAsync(aqs);

            if (devs.Count > 0)
            {
                UsbDevice dev = await UsbDevice.FromIdAsync(
                    devs[0].Id
                    );

                try
                {
                    _fanCtrl = new FanController(dev);
                }
                catch (ArgumentException aex)
                {
                    Console.WriteLine(
                        "Attempted to initialise with \"{0}\".",
                        devs[0].Name
                        );
                    Console.WriteLine("Threw {0}:",
                                      aex.GetType().Name);
                    Console.WriteLine("\t{0}",
                                      aex.Message);
                    Console.WriteLine("Press <ENTER> to retry.");
                    Console.ReadLine();

                    dev.Dispose();

                    return;
                }
            }
            else
            {
                Console.WriteLine("No devices found.");
                Console.WriteLine("Press <ENTER> to recheck.");
                Console.ReadLine();

                return;
            }

            Console.WriteLine("Initialised fan controller.");
            Console.WriteLine("Fans Supported: {0}",
                              _fanCtrl.FansSupported);

            Console.Write("Populating fan information... ");
            _fanCtrl.Update();
            Console.WriteLine("Done.");

            // Set up timer for 1s status refresh
            _updateTimer = new Timer(1000 /* ms */)
            {
                AutoReset = true,
                Enabled = true
            };
            _updateTimer.Elapsed += UpdateHandler;
            // Kick things off by printing the first status.
            Console.WriteLine();
            PrintFanStatus();
            // Hide the cursor
            Console.CursorVisible = false;

            _cursorTopStart = Console.CursorTop;
            Console.Write("\n> ");
            _cursorTop = Console.CursorTop;
            Console.CursorTop--;

            _updateTimer.Start();

            bool keepRunning = true;
            while (keepRunning)
            {
                // Read keys, but don't display automatically. If
                // they were displayed automatically, it would
                // mess with the status reporting.
                var keyInfo = Console.ReadKey(intercept: true);

                keepRunning = ProcessKeys(keyInfo);
            }

            // Clean-up before retry
            _fanCtrl.Dispose();
            _updateTimer.Dispose();
        }


        static void UpdateHandler(object s, ElapsedEventArgs e)
        {
            _fanCtrl.Update();

            // Number of lines to clear to replace the status
            // information we've already printed.
            int linesToClear = _fanCtrl.FansSupported + 1;

            // Start of the first line of status information
            Console.SetCursorPosition(
                left: 0,
                top:  Console.CursorTop - linesToClear
                );

            // Overwrite status information with spaces
            Console.Write(
                new string(
                    ' ', linesToClear * Console.WindowWidth
                    )
                );

            // Return to first line of cleared status information
            Console.SetCursorPosition(
                left: 0,
                top: Console.CursorTop - linesToClear
                );

            // Write out status information
            PrintFanStatus();
        }

        static void PrintFanStatus()
        {
            Console.WriteLine("TIME\t\tFAN\t\tCURRENT");

            foreach (var fs in _fanCtrl.Fans)
            {
                Console.WriteLine(
                    "{0}\t{1}\t\t{2}",
                    fs.AsAtUtc.ToLongTimeString(),
                    fs.Identifier,
                    $"{fs.Current} mA"
                    );
            }
        }

        static StringBuilder _keysRead = new StringBuilder();
        static int _cursorTopStart, _cursorTop;
        static bool ProcessKeys(ConsoleKeyInfo keyInfo)
        {
            // Clears the output line and writes the string.
            void refreshLine(string s)
            {
                // Move to where we left off before
                Console.CursorLeft = 0;
                Console.CursorTop  = _cursorTop;

                // Clear the output line
                Console.Write(
                    new string(' ', Console.WindowWidth)
                    );

                // Reset to start of cleared line
                Console.CursorLeft = 0;
                Console.CursorTop = _cursorTop;

                // Write string to the output
                Console.Write(s);

                // Move to the start line so the status updater
                // doesn't clear the line we've just written
                Console.CursorLeft = 0;
                Console.CursorTop = _cursorTopStart;
            }
            // Writes the specified string and moves down a line
            void writeLine(string s)
            {
                // Move to where we left off
                Console.CursorTop = _cursorTop;
                Console.CursorLeft = 0;

                // Write the string and move to the next line
                Console.WriteLine($"\n{s}\n");

                // Update position tracker
                _cursorTop += 3;

                // Return to where the status updater wants to be
                Console.CursorLeft = 0;
                Console.CursorTop = _cursorTopStart;
            }

            bool retVal = true;

            // If the user backspaces, erase a character
            if (keyInfo.Key == ConsoleKey.Backspace)
            {
                _keysRead.Length = _keysRead.Length - 1;

                refreshLine($"> {_keysRead}");
            }
            // If the key isn't enter...
            else if (keyInfo.Key != ConsoleKey.Enter)
            {
                // If we've reached the end of the buffer, don't
                // process any more keys
                if (_keysRead.Length >= Console.WindowWidth)
                    return retVal;

                // Add it to our record of keys.
                _keysRead.Append(keyInfo.KeyChar);

                refreshLine($"> {_keysRead}");
            }
            // If the enter key is pressed, process what we've
            // read so far.
            else
            {
                const string SET_FAN_MODE = "set";
                const string GET_FAN_MODE = "get";
```

```

                var atoms = _keysRead.ToString()
                                     .Split(' ')
                                     .Where(s => s.Length > 0)
                                     .Select(s => s.ToLower());

                _keysRead.Clear();

                // Do nothing if there is no input.
                if (!atoms.Any())
                    return retVal;

                // Use the first atom as the command.
                switch (atoms.First())
                {
                    /***** Set fan mode *****/
                    case SET_FAN_MODE:
                    {
                        // Parse all parameter atoms.
                        var parsed = atoms.Skip(1) // Skip first
                                          .Select(ParseAtom);

                        // Presence of a null means that one or
                        // more of the parameters is invalid.
                        if (parsed.Any(arg => arg is null))
                        {
                            writeLine("One or more atoms is " +
                                      "invalid.");

                            break;
                        }

                        // We know none are null, so we can get
                        // rid of the Nullable<T> wrapper.
                        var args = parsed.Select(a => a.Value);

                        // Parameters must all be of the voltage
                        // mode, or all of one speed mode.
                        if (args.First().Mode == Mode.Voltage &&
                            args.Any(a => a.Mode != Mode.Voltage)
                            )
                        {
                            writeLine("The first atom specifies "
                                    + "a voltage, but not all " +
                                      "subsequent atoms do.");

                            break;
                        }
                        else if (
                            args.First().Mode != Mode.Voltage &&
                            args.Any(a => a.Mode == Mode.Voltage)
```

```
          )
      {
          writeLine("The first atom specifies "
                    + "a speed, but not all " +
                      "subsequent atoms do.");

          break;
      }

      // Having verified that all arguments
      // are valid, use the value of the first
      // to determine which mode to set.
      if (args.First().Mode == Mode.Voltage)
      {
          bool worked;
          try
          {
              var points = args.Select(
                  a => ((double)a.Qty, a.Tmp)
                  );

              worked = _fanCtrl.SetVoltageMode(
                  _fanCtrl.Fans[0], points
                  );

              if (worked)
              {
                  writeLine(
                      "Fan configuration " +
                      "updated."
                      );
              }
              else
              {
                  writeLine(
                      "Fan configuration " +
                      "failed."
                      );
              }
          }
          catch (Exception e) when (
              e is ArgumentException ||
              e is ArgumentOutOfRangeException
              )
          {
              writeLine(e.Message);
              worked = false;
          }
      }
      else
      {
```

```
                            writeLine(
                                "Configuration command " +
                                "acknowledged, parameters " +
                                "verified, not implemented."
                                );
                        }

                    } break;

                    /***** Get fan mode *****/
                    case GET_FAN_MODE:
                    {
                        var points = _fanCtrl.GetMode(
                            _fanCtrl.Fans[0]
                            );

                        int i = 1;
                        foreach (var (V, T) in points)
                        {
                            writeLine(
                                $"Point #{i++}: {V}V at {T}C"
                                );
                        }
                    } break;

                    /***** Unrecognised command *****/
                    default:
                    {
                        writeLine("Unrecognised command.");
                    } break;
                }

                refreshLine("> ");
            }

            return retVal;
        }

        // Possible modes represented in mode data.
        enum Mode { Voltage, SpeedPercent, SpeedRpm };
        // Attempts to parse an atom.
        static (Mode Mode, int Qty, int Tmp)? ParseAtom(string a)
        {
            // ASCII/UTF-8 number codepoint range is 0x30-0x39.
            bool isNumeric(char c) => c >= 0x30 && c <= 0x39;

            // The following is not the most efficient way to
            // retrieve each part of the atom, but it is probably
            // one of the easiest ways, and so is good enough for
            // demonstration purposes.
            //
            // The controlled quantity comes first, and is
            // entirely numeric.
            var strQuantity = a.TakeWhile(isNumeric);
            // The controlled quantity is followed by the unit,
            // specifying what the quantity is.
            var strUnit = a.SkipWhile(isNumeric)
                           .TakeWhile(c => !isNumeric(c));
            // Which is immediately followed by the temperature.
            var strTemp = a.SkipWhile(isNumeric)
                           .SkipWhile(c => !isNumeric(c))
                           .TakeWhile(isNumeric);
            // Which is itself immediately followed by a unit of
            // temperature, which will always be C here.
            var strTempUnit = a.SkipWhile(isNumeric)
                               .SkipWhile(c => !isNumeric(c))
                               .SkipWhile(isNumeric)
                               .TakeWhile(c => !isNumeric(c));

            // Because we included the '@' symbol as a separator,
            // each of the units will have it at the end (if they
            // are correct).
            Mode mode;
            switch (new string(strUnit.ToArray()))
            {
                case "v@":   mode = Mode.Voltage;       break;
                case "%@":   mode = Mode.SpeedPercent;  break;
                case "rpm@": mode = Mode.SpeedRpm;      break;

                // If we don't recognise the unit, then we can't
                // continue with parsing the atom.
                default:     return null;
            }

            // Knowing the unit, we can now enforce the limits on
            // values imposed by the Appendix D protocol.
            int quantity, temp;
            // To do this, the strings must be parsed to numbers,
            // and we must fail if they are not valid numbers.
            if (!int.TryParse(new string(strQuantity.ToArray()),
                                         out quantity) ||
                !int.TryParse(new string(strTemp.ToArray()),
                              out temp))
            {
                return null;
            }

            // A temperature must be in the range 0-127C.
            if (temp < 0 || temp > 127)
                return null;

            // The validity of the controlled quantity depends
            // on the mode specified.
            bool valQty;
            switch (mode)
            {
                // The voltage is 0-12V. Technically it should be
                // in ~47mV increments, but this simplifies the
                // demonstration code a little.
                case Mode.Voltage:
                    valQty = quantity > 0 && quantity <= 12;
                    break;

                // The percentage must be 0-100%.
                case Mode.SpeedPercent:
                    valQty = quantity > 0 && quantity <= 100;
                    break;

                // The RPM must be 0 to 2**15 - 1.
                case Mode.SpeedRpm:
                    valQty = quantity > 0 &&
                             quantity <= (2 << 15) - 1;
                    break;

                // This will never happen, but the compiler
                // complains since it technically means the
                // variable could remain uninitialised.
                default: valQty = false; break;
            }

            // We can't continue if the quantity is invalid.
            if (!valQty)
                return null;

            // Successful parse.
            return (mode, quantity, temp);
        }
    }
}
```

FanController.cs

An abstraction over the USB interface to the fan controller which implements the protocol contained in Appendix D.

```
/* Author:  Liam McSherry
 * Date:    3rd March 2018
 * Notes:   Represents a USB-connected fan controller.
 */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Timers;
using System.IO;

using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Storage.Streams;
using Windows.Devices.Usb;

namespace McSherry.Edu.HND.UsbProto
{
    using TransferType = UsbControlTransferType;

    /// <summary>
    /// <para>
    /// Represents a USB-connected fan controller.
    /// </para>
    /// </summary>
    public sealed class FanController
        : IDisposable
    {
        // The USB Vendor and Product IDs of controllers we'll
        // attempt to recognise as valid.
        private const uint VendorID = 0x1209;
        private const uint ProductID = 0x0001;

        // The descriptor type and size for the fan controller
        // configuration descriptor.
        private const byte FccDescriptor = 0x20;
        private const byte FccDescriptorSize = 8;
        // The dummy version value we're reporting.
        private const uint FccVersion = 0x0011;

        /// <summary>
        /// <para>
        /// Parses a fan controller configuration descriptor from
        /// a provided <see cref="UsbDescriptor"/>.
        /// </para>
        /// </summary>
        /// <param name="desc">
        /// The <see cref="UsbDescriptor"/> to parse.
        /// </param>
        /// <returns>
        /// A tuple containing the number of fans supported by
        /// the controller and the specification version the
        /// controller conforms to.
        /// </returns>
        /// <exception cref="FormatException">
        /// The <see cref="UsbDescriptor"/> was not in a valid
        /// format for a fan controller configuration descriptor.
        /// </exception>
        private static (int fans, uint version) _parseFccDesc(
            UsbDescriptor desc
            )
        {
            // Field positions
            const int bLength         = 0,
                      bDescriptorType = 1,
                      bmAttributes    = 2,
                      bcdVersion      = 3;


            int fans;
            uint version;
            var buf = new byte[8];

            desc.ReadDescriptorBuffer(buf.AsBuffer());

            // Basic verification: length and type
            if (buf[bLength] != FccDescriptorSize ||
                buf[bDescriptorType] != FccDescriptor)
            {
                throw new FormatException(
                    "The provided descriptor did not specify " +
                    "the correct length and/or descriptor type."
                    );
            }

            // The number of fans supported is stored in the
            // lowest four bits, and is zero-based.
            fans = (int)(buf[bmAttributes] & 0x0F) + 1;

            // The version is stored as two bytes, in little-
            // -endian order.
            version = buf[bcdVersion] |
                      (uint)(buf[bcdVersion + 1] << 8);

            return (fans, version);
        }


        // The fan controller USB device.
        private readonly UsbDevice _dev;
        // List of fan statuses.
        private readonly List<FanStatus> _fans;


        /// <summary>
        /// <para>
        /// Creates a new <see cref="FanController"/> from a
        /// provided <see cref="UsbDevice"/> instance.
        /// </para>
        /// </summary>
        /// <param name="device">
        /// The <see cref="UsbDevice"/> from which to create the
        /// new instance.
        /// </param>
        /// <exception cref="ArgumentException">
        /// <paramref name="device"/> is not recognised as a
        /// valid fan controller.
        /// </exception>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <paramref name="device"/> reports a fan controller
        /// specification version which is not supported.
        /// </exception>
        /// <exception cref="ArgumentNullException">
        /// <paramref name="device"/> is null.
        /// </exception>
        public FanController(UsbDevice device)
        {
            // ***** Null check
            if (device == null)
            {
                throw new ArgumentNullException(
                    paramName: nameof(device)
                    );
            }

            // ***** Valid fan controller checks
            //
            // Device reports the anticipated VID/PID pair
            if (device.DeviceDescriptor.VendorId != VendorID ||
                device.DeviceDescriptor.ProductId != ProductID)
            {
                throw new ArgumentException(
                    "The USB device did not report the expected "
                  + "Vendor and Product Identifiers."
                    );
            }

            // Device reports the anticipated config. descriptor.
            var fccDesc =
               (from d in device.Configuration.Descriptors
                where d.DescriptorType == FccDescriptor &&
                      d.Length == FccDescriptorSize
                select d).FirstOrDefault();
            // Null means no results
            if (fccDesc == null)
            {
                throw new ArgumentException(
                    "The USB device did not report a fan " +
                    "controller configuration descriptor."
                    );
            }

            // Attempt to parse the descriptor.
            uint version;
            try
            {
                (this.FansSupported, version) = _parseFccDesc(
                    fccDesc
                    );
            }
            // Report failure.
            catch (FormatException fex)
            {
                throw new ArgumentException(
                    "The USB device reported a fan controller " +
                    "configuration descriptor which was in an " +
                    "invalid format.",
                    fex
                    );
            }

            // Device reports a supported version
            if (version != FccVersion)
            {
                throw new ArgumentOutOfRangeException(
                    "The USB device reported a fan controller " +
                    "specification version which was not " +
                    "supported."
                    );
            }

            _dev = device;
            _fans = new List<FanStatus>((int)this.FansSupported);
        }


        /// <summary>
        /// The number of fans the controller supports.
        /// </summary>
        public int FansSupported { get; }

        /// <summary>
        /// <para>
        /// Status information for all supported fans.
        /// </para>
        /// </summary>
        public IReadOnlyList<FanStatus> Fans => _fans;


        /// <summary>
        /// <para>
        /// Retrieves the current status of connected fans and
        /// updates the contents of <see cref="Fans"/>.
        /// </para>
        /// </summary>
        public void Update()
        {
            var usbSetup = new UsbSetupPacket
            {
                // Device-to-host, class-specific, device request
                RequestType = new UsbControlRequestType
                {
                    Direction = UsbTransferDirection.In,
                    ControlTransferType = TransferType.Class,
                    Recipient = UsbControlRecipient.Device
                },

                // Request 0 (GET_FAN_ID)
                Request = 0,

                // Default value to zero, it'll actually be used
                // to specify the fan we want
                Value = 0,

                // Would usually specify the receiving interface,
                // but our USB device doesn't care.
                Index = 0,

                // GET_FAN_ID responses are 8 bytes long.
                Length = 8,
            };

            // Query for each of the fans supported.
            for (ushort i = 0; i < this.FansSupported; i++)
            {
                // Adjust our SETUP packet so we specify that we
                // want to query the next fan. We don't need to
                // worry about the upper byte, as we'll never
                // exceed 255 fans.
                usbSetup.Value = i;

                // Buffer for the returned result
                var resBuf = new byte[8];

                // Send the request and retrieve an awaitable
                var task = _dev.SendControlInTransferAsync(
                    usbSetup, resBuf.AsBuffer()
                    );

                // Wait for completion. This wouldn't be suitable
                // in real code as we'd want to make sure there
                // was no possibility of infinitely looping here,
                // but we can work around such errors in testing.
                while (task.Status == AsyncStatus.Started)
                    continue;

                // We'd also want to report a real error message
                // here if this were real code.
                if (task.Status != AsyncStatus.Completed)
                {
                    throw new InvalidOperationException(
                        "Error in transfer."
                        );
                }

                // Parse the buffer into a [FanStatus].
                var fs = FanStatus.Parse(resBuf.ToArray());

                // Assign the [FanStatus] to the list, creating
                // the space necessary if the list is empty.
                if (_fans.Count == 0)
                {
                    _fans.Add(fs);
                }
                else
                {
                    _fans[i] = fs;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Configures the fan controller to adjust the speed of
        /// the specified fan in accordance with the provided set
        /// of points.
        /// </para>
        /// </summary>
        /// <param name="fan">
        /// The fan for which to set the control configuration.
        /// </param>
        /// <param name="points">
        /// A set of points determining the voltage supplied to
        /// the fan at specified temperatures.
        /// </param>
        /// <returns>
        /// True if the fan is successfully configured, false if
        /// otherwise.
        /// </returns>
        /// <exception cref="ArgumentNullException">
        /// <paramref name="points"/> is null.
        /// </exception>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <paramref name="points"/> contains a point with an
        /// invalid voltage or temperature.
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <paramref name="fan"/> does not specify a valid fan.
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// Voltage control is not supported for the specified
        /// fan.
        /// </exception>
        public bool SetVoltageMode(
            FanStatus fan,
            IEnumerable<(double Voltage, int Temperature)> points
            )
        {
            // The voltage is provided to the controller as a
            // multiple of ~47.06mV, so any absolute value given
            // here must be converted to this format.
            byte normaliseVolts(double volts)
                => (byte)Math.Round(volts / 47.06E-3);

            if (points == null)
            {
                throw new ArgumentNullException(
                    "The provided set of points is null."
                    );
            }

            if (!_fans.Any(f => f.Identifier == fan.Identifier))
            {
                throw new ArgumentException(
                    "The specified fan is not valid."
                    );
            }

            if (!fan.ControlModes.Voltage)
            {
                throw new InvalidOperationException(
                    "The specified fan does not support " +
                    "voltage control."
                    );
            }

            // Voltage must be in the range 0-12V.
            if (points.Any(p => p.Voltage < 0 || p.Voltage > 12))
            {
                throw new ArgumentOutOfRangeException(
                    "The voltage specified for one of the " +
                    "provided points was not between 0 and 12."
                    );
            }

            // Temperature must be in the range 0-127C.
            if (points.Any(p => p.Temperature < 0 ||
                                p.Temperature > 127))
            {
                throw new ArgumentOutOfRangeException(
                    "The temperature specified for one of the " +
                    "provided points was not between 0 and 127."
                    );
            }


            // Buffer for containing mode data to be transmitted
            // to the device. Each voltage control entry is two
            // bytes long.
            var ms = new MemoryStream();

            // Each point must be serialised.
            foreach (var (Voltage, Temperature) in points)
            {
                // First byte is the temperature, with the zeroth
                // bit indicating whether this is the end.
                ms.WriteByte((byte)(Temperature << 1));
                // Second byte is the voltage level.
                ms.WriteByte(normaliseVolts(Voltage));
            }

            // For the last byte, we want to set the END bit.
            var bufArr = ms.ToArray();
            bufArr[bufArr.Length - 2] |= 1;

            ms.Dispose();

            var usbSetup = new UsbSetupPacket
            {
                RequestType = new UsbControlRequestType
                {
                    Direction = UsbTransferDirection.Out,
                    ControlTransferType = TransferType.Class,
                    Recipient = UsbControlRecipient.Device,
                },

                // SET_FAN_MODE is 0x01
                Request = 0x01,

                Index = 0,

                // Each entry is two bytes long.
                Length = (uint)points.Count() * 2,

                // See Appendix D2.4.3 (Table 4), the value is a
                // bit field containing information about the
                // request. Mostly reserved.
                Value = (uint)(_fans.IndexOf(fan) & 0xF) |
                              (0b00U << 4)
            };

            var ba = bufArr.AsBuffer();

            var task = _dev.SendControlOutTransferAsync(
                usbSetup, bufArr.AsBuffer()
                );

            // Wait until completion
            while (task.Status == AsyncStatus.Started)
                continue;

            // Indicate failure if the task fails
            if (task.Status != AsyncStatus.Completed)
                return false;

            // If it doesn't fail, it succeeds.
            return true;
        }

        /// <summary>
        /// Retrieves the current configuration of the fan
        /// controller as a set of points.
        /// </summary>
        /// <param name="fan">
        /// The fan the configuration of which to retrieve.
        /// </param>
        /// <returns>
        /// A set of points representing the configuration of
        /// the fan.
        /// </returns>
        /// <exception cref="ArgumentException">
        /// <paramref name="fan"/> does not specify a valid fan.
        /// </exception>
        public IEnumerable<(double Voltage, int Temperature)>
            GetMode(FanStatus fan)
        {
            if (!_fans.Any(f => f.Identifier == fan.Identifier))
            {
                throw new ArgumentException(
                    "The specified fan is not valid."
                    );
            }
```

```

                // We're only handling the case of voltage mode data
                // here, but in a real application there it would be
                // possible to receive speed mode data.
                //
                // We first need to retrieve the data.
                var usbSetup = new UsbSetupPacket
                {
                    RequestType = new UsbControlRequestType
                    {
                        Direction = UsbTransferDirection.In,
                        ControlTransferType = TransferType.Class,
                        Recipient = UsbControlRecipient.Device
                    },

                    // Request 2 (GET_FAN_MODE)
                    Request = 2,

                    // The value specifies the fan we want
                    Value = (uint)_fans.FindIndex(
                        f => f.Identifier == fan.Identifier
                        ),

                    // Normally specifies the interface, but our
                    // device doesn't care so this is ignored.
                    Index = 0,

                    // We don't know the length ahead of time, so
                    // we specify the maximum length.
                    Length = 32
                };

                // We know this is the maximum size that the device
                // can return, but in the real world we'd have some
                // indicator of total length (which wasn't included
                // in the Appendix D protocol).
                var buf = new byte[32];

                var task = _dev.SendControlInTransferAsync(
                    usbSetup, buf.AsBuffer()
                    );

                // Wait for completion
                while (task.Status == AsyncStatus.Started)
                    continue;

                if (task.Status != AsyncStatus.Completed)
                {
                    throw new InvalidOperationException(
                        "Error in transfer."
                        );
```

```
            }

            // We now deserialise the points. They're in the same
            // format as when we sent them, so it's simple.
            var points = new List<(double, int)>();
            for (int i = 0; i < buf.Length; i += 2)
            {
                // Break if the END bit is set
                if ((buf[i] & 1) > 0)
                    break;

                points.Add((
                    // The temperature field is the upper seven
                    // bits of the first byte of the entry.
                    (buf[i] & 0b1111_1110) >> 1,
                    // And the voltage field is the second byte
                    // of the entry, representing the 0-12V range
                    // as portions of 1/255.
                    (buf[i + 1] * 12) / 255
                    ));
            }

            return points.AsReadOnly();
        }

        /// <summary>
        /// <para>
        /// Disposes resources held by the instance.
        /// </para>
        /// </summary>
        public void Dispose()
        {
            _dev.Dispose();
            _fans.Clear();
        }
    }

    /// <summary>
    /// <para>
    /// Represents the status at a particular point in time of a
    /// fan connected to the controller.
    /// </para>
    /// </summary>
    public struct FanStatus
    {
        /// <summary>
        /// <para>
        /// Parses a <see cref="FanStatus"/> from a byte array.
        /// </para>
        /// </summary>
        /// <param name="fanId">
        /// An array containing the response from the controller
        /// when fan identification was requested.
        /// </param>
        /// <returns>
        /// The <see cref="FanStatus"/> represented by the bytes
        /// in the byte array.
        /// </returns>
        /// <exception cref="ArgumentNullException">
        /// <paramref name="fanId"/> is null.
        /// </exception>
        /// <exception cref="FormatException">
        /// <paramref name="fanId"/> does not represent a valid
        /// fan identification response.
        /// </exception>
        public static FanStatus Parse(byte[] fanId)
        {
            const int FanIdLength = 8;
            // Field indices
            const int bmAttributes  = 0;
            const int wCurrent      = 3;
            const int iIdentifier   = 5;

            // Null check
            if (fanId == null)
            {
                throw new ArgumentNullException(
                    "The provided byte array was null."
                    );
            }

            // Minimum length check
            if (fanId.Length < FanIdLength)
            {
                throw new FormatException(
                    "The provided byte array was not long " +
                    "enough to contain valid data."
                    );
            }

            // We're treating now as when the measurement was
            // taken, just because it's a bit simpler and makes
            // no real impact.
            var ts = DateTime.UtcNow;


            // bmAttributes is a three-byte field, provided in
            // little-endian byte order.
            uint attrs =        fanId[bmAttributes]             |
                         ((uint)fanId[bmAttributes + 1] <<  8) |
                         ((uint)fanId[bmAttributes + 2] << 16) ;


            // Voltage and speed control flags
            (bool S, bool V) = ((attrs & 0b10) > 0,  // Bit 2
                                (attrs & 0b01) > 0); // Bit 1

            double startV;
            // If voltage control is supported, then the starting
            // voltage is the value in bmAttributes[6:2], which
            // represents steps of 375mV starting at 0 = 375mV.
            if (V)
                startV = 0.375 * (((attrs & 0b1111100) >> 2) + 1);
            // If it isn't supported, there is no valid value.
            else
                startV = Double.NaN;

            double minSpeed;
            uint maxRpm;
            // If speed control is supported, then the minimum
            // supported speed (as a percentage of maximum) is
            // the value given in bmAttributes[11:7]. That value
            // is an integer 0..31, so we divide by 10.
            //
            // Likewise, maximum RPM is in bmAttributes[23:12].
            if (S)
            {
                minSpeed = ((attrs & 0b11111000000) >> 6) / 10d;
                maxRpm = (attrs & 0xFFF800) >> 11;
            }
            // If it isn't supported, there is no valid value
            // for either.
            else
            {
                minSpeed = Double.NaN;
                maxRpm = 0;
            }


            // The fan current is an integer stored in bytes at
            // offsets 3 and 4, little-endian order.
            uint current =       fanId[wCurrent]           |
                           ((uint)fanId[wCurrent + 1] << 8) ;


            // String index for the fan's unique identifier. This
            // will do, for now.
            uint idIndex = fanId[iIdentifier];

            return new FanStatus(
                ts, idIndex.ToString(), (S, V), startV,
                minSpeed, maxRpm, current
                );
        }

        // The date and time this measurement was taken, in UTC.
        private readonly DateTime           _ts;
        // A string identifying the fan.
        private readonly string             _id;
        // The control modes supported by the fan.
        private readonly (bool S, bool V)   _ctrl;
        // The voltage at which the fan will start.
        private readonly double             _startVolts;
        // The minimum speed % if the fan supports speed control.
        private readonly double             _minSpeedPc;
        // The maximum speed of the fan in RPM.
        private readonly uint               _maxSpeed;
        // The current drawn by the fan.
        private readonly uint               _fanCurrent;

        internal FanStatus(
            DateTime                    ts,
            string                      id,
            (bool Speed, bool Voltage)  control,
            double                      startVolts,
            double                      minSpeedPc,
            uint                        maxSpeed,
            uint                        fanCurrent
            )
        {
            _ts         = ts;
            _id         = id;
            _ctrl       = control;
            _startVolts = startVolts;
            _minSpeedPc = minSpeedPc;
            _maxSpeed   = maxSpeed;
            _fanCurrent = fanCurrent;
        }

        /// <summary>
        /// <para>
        /// The date and time this measurement was recorded (in
        /// UTC). That is, the fan "as at" this time.
        /// </para>
        /// </summary>
        public DateTime AsAtUtc => _ts;

        /// <summary>
        /// <para>
        /// A string uniquely identifying the fan.
        /// </para>
        /// </summary>
        public string Identifier => _id;

        /// <summary>
        /// <para>
        /// The modes of control supported for the fan.
        /// </para>
        /// </summary>
        public (bool Voltage, bool Speed) ControlModes => _ctrl;

        /// <summary>
        /// <para>
        /// Where voltage control is supported, the voltage at
        /// which the fan will start. Otherwise, NaN.
        /// </para>
        /// </summary>
        public double StartVoltage => _startVolts;
        /// <summary>
        /// <para>
        /// Where speed control is supported, the minimum speed
        /// configurable for the fan as a percentage of total
        /// speed. Otherwise, NaN.
        /// </para>
        /// </summary>
        public double MinimumSpeed => _minSpeedPc;

        /// <summary>
        /// <para>
        /// The maximum speed of the fan in RPM. If the maximum
        /// speed cannot be determined, zero.
        /// </para>
        /// </summary>
        public uint MaximumSpeed => _maxSpeed;

        /// <summary>
        /// <para>
        /// The current, in milliamps, that the fan is currently
        /// drawing.
        /// </para>
        /// </summary>
        public uint Current => _fanCurrent;
    }
}
```

C5 PWM DAC High-Z Response Plot

The following is the source code for the script, written in R, used to plot the figure showing the PWM DAC response under high-impedance conditions used in Appendix F7.8.

```r
Data_48R <- read.table("PWM-DAC-OpenCircuit-Sim-48R", header=TRUE)
Data_1M  <- read.table("PWM-DAC-OpenCircuit-Sim-1M", header=TRUE)
Time_48R <- Data_48R[,1]
Time_1M  <- Data_1M[,1]
V_48R    <- Data_48R[,2]
V_1M     <- Data_1M[,2]

# Seconds to milliseconds
Time_48R <- Time_48R * 1000
Time_1M  <- Time_1M  * 1000

# Plot 48R voltage with grid
plot(Time_48R, V_48R, type="l",
     xlab="Time (ms)", xlim=c(0,10),
     ylab="Voltage (V)", ylim=c(0,14),
     lab=c(5, 7, 7), lwd=2, col="blue")

grid()

# Plot 1M voltage
points(Time_1M, V_1M, type="l",
       lwd=2, col="red")

# Title etc.
title("PWM DAC Response under High-Z Conditions")

legend("bottomright", bg="white",
       legend=c("1 MΩ", "48 Ω"), pch=15,
       col=c("red", "blue"), text.col=c("red", "blue"))

text(x=8.1, y=c(12.05, 7.25) + 0.75, labels=c("1 MΩ", "48 Ω"),
     adj=0, col=c("red", "blue"))
```

• PAGE INTENTIONALLY LEFT BLANK •

Appendix D
Fan controller device class specification

D1 Introduction

This appendix provides a specification for the protocol used in communicating with the fan controller over USB. Where possible, this appendix has been written in the style of a device class specification and following the guidelines in the USB Common Class Specification (USB-IF, 1997).

While ideally a generic specification for fan controller-type devices would be produced, constraints on time and consequential constraints on complexity mean that this is likely to be infeasible. Taking this into consideration, genericity has been maintained where practical.

D1.1 Scope

This device class specification is intended to include devices with largely similar capabilities to the device described by the main body of the report to which this specification is an appendix. In particular, a device is included irrespective of:

- The number of computer fans supported.
- Of the three principal varieties identified in the main body of the report, the particular combination of the varieties of computer fan supported.

D1.2 Related Documents

Bradner, S., 1997. RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. [Online]
Available at: https://tools.ietf.org/html/rfc2119

USB-IF, 1997. Universal Serial Bus Common Class Specification. rev. 1.0 ed. s.l.:Universal Serial Bus Implementers Forum.

USB-IF, 2000. Universal Serial Bus Specification. rev. 2.0 ed. s.l.:Universal Serial Bus Implementers Forum.

D1.3 Terminology

In this appendix, the words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* are to be interpreted as described in RFC 2119 (Bradner, 1997), unless used in a section marked informative.

Other terms are to be taken as having the same meaning as they have in the USB specification (USB-IF, 2000), unless context requires otherwise or derogation is expressly made.

D2 Functional characteristics

D2.1 Operational model

A fan controller is largely a “fire and forget” device—once configured by the host computer’s human operator, it requires minimal interaction with that operator or with any automated process on the host computer. Excepting meaningful changes in circumstances, it is not expected that any fan controller setting will be changed after the fan controller’s initial setting up.

Accordingly, there are two broad groups of operation that can be performed on a fan controller—operations for status monitoring (whether for the controller or for the fans attached to it) and operations for control (although given the nature of the fan controller, the greatest portion of the control operations are settings-changing operations rather than operations which directly control a fan).

D2.2 Interfaces

A fan controller must support a single interface using the Default Control Pipe.

D2.3 Descriptors

A fan controller supports both the standard USB descriptors and a number of class-specific descriptors. A fan controller must implement standard descriptors as they are defined in the USB specification (pp. 261–274).

A fan controller must support the following class-specific descriptor. Per the USB specification (pp. 260–261), a fan controller must return this descriptor after the relevant configuration descriptor in the GetDescriptor(Configuration) response.

Table 1

Class-specific Fan Controller Configuration Descriptor

Offset	Field	Size	Value	Description
0	`bLength`	1	Number	Size of this descriptor in bytes.
1	`bDescriptorType`	1	`20h`	FAN_CTRLR descriptor.
2	`bmAttributes`	1	Bitmask	Fan controller attributes. D7...4: Reserved. D3...0: One less than the maximum number of fans supported.
3	`bcdVersion`	2	BCD	Fan controller device class specification version number, e.g. `0105h` = v1.05.
5	–	3	—	Reserved.

D2.4 Requests

A fan controller supports both the standard USB device requests and a number of class-specific requests. A fan controller must implement the standard device requests defined in the USB specification (pp. 250–260) as they are defined by that specification.

A fan controller must support the class-specific requests given in Table 2.

Table 2

Class-specific requests

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**	**Data**
`IN CLASS INTERFACE`	`GET_FAN_STATUS (0)`	Fan Selector	Interface	8	Byte set
`OUT CLASS INTERFACE`	`SET_FAN_MODE (1)`	Fan Selector	Interface	Depends on `wValue`	Depends on `wValue`
`IN CLASS INTERFACE`	`GET_FAN_MODE (2)`	Fan selector	Interface	Length of fan mode data	Fan mode, as set by `SET_FAN_MODE`

D2.4.1 GET_FAN_ID (bRequest = 0)

This request retrieves attributes and status information for a fan connected to the controller.

The value in `wValue[3:0]` is a zero-based fan selector, with a value in the range of 0 to `FAN_CTRLR.bmAttributes[3:0]`. The remainder of `wValue` is reserved and must be reset to zero.

The data returned by this request must be in the format described by Table 3.

Table 3
GET_FAN_ID response format

Offset	Field	Size	Value	Description
0	`bmAttributes`	3	Bitmap	The attributes for the fan. D23...D12: Maximum rpm If D01 = 1, the maximum rpm of the fan. If D01 = 0, zero. D11...D07: Minimum speed If D01 = 1, the minimum speed in the range 0–31%. If D01 = 0, zero. D06...D02: Starting voltage If D00 = 1, the minimum starting voltage of the fan, in 375 mV increments (e.g. 0 = 375 mV, 1 = 750 mV) If D00 = 0, zero. D01: Speed control D00: Voltage control
3	`wCurrent`	2	Word	The current, in milliamps, that the fan is currently drawing.
5	`iIdentifier`	2	Index	The index of a string identifier which uniquely identifies the fan.
7	–	1	—	Reserved.

If the fan specified in `wValue` does not exist, a fan controller must respond with a request error.

D2.4.2 SET_FAN_MODE (bRequest = 1)

This request adjusts how a fan controller controls a fan connected to it.

The format of `wValue` is given in Table 4.

Table 4
SET_FAN_MODE request, `wValue` format

D15	D14	D13	D12	D11	D10	D9	D8
Reserved							
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Reserved		Mode selector		Fan selector			

The fan selector value in `wValue[3:0]` is a zero-based fan selector, with a value in the range of 0 to `FAN_CTRLR.bmAttributes[3:0]`.

The mode selector in `wValue[5:4]` determines the content of the data payload sent with the request. Valid values are given in Table 5.

Table 5

SET_FAN_MODE request, valid mode selectors

11b	10b	01b	00b
Reserved	Reserved	Speed	Voltage

The *voltage* mode selector is only valid if voltage control is specified in the response to a `GET_FAN_ID` request for the specified fan. The data payload of the request is in the format specified in section D3.1.

The *speed* selector is only valid if speed control is specified in the response to a `GET_FAN_ID` request for the specified fan. The data payload of the request is in the format specified in section D3.2.

A fan controller must respond with a request error if:

- The fan specified in `wValue` does not exist.
- The mode selector is not valid, either by virtue of the attributes for the fan or because it is reserved or unspecified.
- The data payload for the request is not in a valid format.

A fan controller must store and retain (including over power cycles), for each fan, the latest mode and mode data specified through this request. This retained data should be used as the initial state of the controller after power-on, but a fan controller may provide means (such as by a jumper or switch) of selecting another initial state.

D2.4.3 GET_FAN_MODE (bRequest = 2)

This request retrieves the current mode and mode data for a fan connected to the controller.

The value in `wValue[3:0]` is a zero-based fan selector, with a value in the range of 0 to `FAN_CTRLR.bmAttributes[3:0]`. The remainder of `wValue` is reserved and must be reset to zero.

The data returned by the request is a byte followed by the mode data equivalent to that provided to the `SET_FAN_MODE` request. The two least-significant bits of the first byte are a mode selector (see Table 5), and the 6 most-significant bits are reserved and must be reset to zero.

A fan controller must respond with a request error if the fan specified in `wValue` does not exist.

D3 Mode data format

The voltage- and speed-controlled modes are specified as points on a curve or line which maps temperature to a voltage or speed. A similar, but not identical, format is used for each control mode.

Where the temperature is between two points, a fan controller should use the value specified for the lowest-temperature point of the two. A fan controller may implement a tolerance for points based on the accuracy and precision of its temperature sensor—for example, given a sensor with ±0.5 °C accuracy and a point set at 40 °C, a fan controller may consider any temperature from 39.5–40.5 °C to be on the 40 °C point.

D3.1 Voltage control mode format

This format is a sequence of two-byte entries, each as in Table 6.

Table 6

Voltage control mode entry format

D15	D14	D13	D12	D11	D10	D9	D8
Voltage level							
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Temperature							END

The END field indicates whether this entry is the last in the sequence when set. If this bit is not set, a host must provide a further entry. A host should not include additional data after the last entry.

The temperature field is an unsigned integer, the value of which is a temperature in the range 0–127 °C.

The voltage level represents a voltage in the range 0–12 V, in $^1/_{255}$ V increments. A fan controller must stop the fan if the value specified is less than the starting voltage given in `GET_FAN_ID.bmAttributes[6:2]`.

D3.2 Speed control mode format

This format is a sequence of variable-length, two- or three-byte entries, each in the format given in Table 7.

Table 7

Speed control mode entry format

D23	D22	D21	D20	D19	D18	D17	D16
Speed (B)							
D15	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**
Speed (A)							TYPE
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Temperature							END

The END field indicates whether this entry is the last in the sequence when set. If this bit is not set, a host must provide a further entry. A host should not include additional data after the last entry.

The temperature field is an unsigned integer, the value of which is a temperature in the range 0–127 °C.

The TYPE field indicates the length of the entry and how to interpret the speed fields. If the TYPE field is set, the entry is three bytes long and the Speed (A) and the Speed (B) field form a 15-bit unsigned integer giving the desired fan speed in revolutions per minute (rpm). The Speed (A) field is the less-significant field. The specified speed must not exceed the *maximum rpm* value given in the response to `GET_FAN_ID` for the fan.

If the TYPE field is unset, the entry is two bytes long and the Speed (A) field is an unsigned integer giving a fan speed in the range 0–100%. A fan controller must consider values in the range 101–127 invalid.

Note: The more complex option has been assigned TYPE = 1 to simplify the process of reading entries in software. The Speed (A) field can be read unconditionally into a 16-bit unsigned integer, with the Speed (B) field conditionally loaded into the same unsigned integer. Only the loading of Speed (B) need be inside a conditional (e.g. `if`) statement.

If the specified speed is 0 rpm or 0%, a fan controller must stop the fan. If the specified speed—whether in rpm or a percentage—is equivalent to a speed less than that given in the *minimum speed* field in the `GET_FAN_ID` response for the fan, a fan controller should operate the fan at its specified minimum speed.

Appendix E
Summary of expenditure

This appendix gives an exhaustive list of all spending made in pursuance of the objectives and requirements. Spending in currencies other than pound sterling has been, where possible, converted at the conversion rate available on the day of the spend.

13th November 2017

Description	Spend
Proof-of-concept prototype circuit: manufacture, assembly, and postage; associated bank processing fees.	£ 83.05
Subtotal	£ 83.05
Running total	£ 83.05

28th November 2017

Description	Spend
Fractal Design Silent Series R3 (1600 rpm, 3-pin, 0.06 A)	£ 5.28
Corsair Air Series SP120 PWM (2350 rpm, 4-pin, 0.25 A) ×2	£ 12.24
Subtotal	£ 17.52
Running total	£ 100.57

21st December 2017

Description	Spend
Proof-of-concept prototype circuit: VAT, customs duty, delivery service processing fees.	£ 32.97
Subtotal	£ 32.97
Running total	£ 133.54

9th April 2018

Description	Spend
Report: printing and binding, three copies.	£ 75.00
Report: postage.	£ 6.50
Subtotal	£ 81.50
Running total	£ 215.04

Appendix F
Proof-of-concept prototype

F1 Bill of Materials

The Bill of Materials for the proof-of-concept prototype is as follows. Parts with an asterisk mark (*) are generic parts stocked by the manufacturer, MacroFab.

The manufacturer's pricing was in US dollars, and so this Bill of Materials includes prices in US dollars rather than pound sterling.

Designators	**Part No.**	**Price (×1)**	**Price (all)**
C1, C2, C4, C5	*MF-CAP-0603-0.1uF	$ 0.050	$ 0.200
C3	Taiyo Yuden UMK316AB7475KL-T	$ 0.295	$ 0.295
D1	ComChip CDBA540-HF	$ 0.543	$ 0.543
IC1	TI CD74HC4050M96	$ 1.010	$ 1.010
IC2	NXP PCT2075TP,147	$ 0.802	$ 0.802
IC3, IC4	TI INA180A2IDBVT	$ 0.673	$1.346
JP1	Sullins SFH11-PBPC-D10-RA-BK	$ 1.652	$ 1.652
JP2	Altech MBES-152	$ 0.260	$ 0.260
JP3, JP4	Molex 47053-1000	$ 0.519	$ 1.038
L1	Bourns PM2120-471K-RC	$ 2.832	$ 2.832
M1, M2, M3, M4	Fairchild FDMS7682	$ 0.496	$ 1.982
R1, R2, R9, R13	*MF-RES-0805-4.7K	$ 0.050	$ 0.200
R3, R10	Vishay CRCW080538K3FKEB	$ 0.118	$ 0.236
R4	Vishay CRCW12101K10FKEAHP	$ 0.118	$ 0.118
R5, R11	*MF-RES-0805-10K	$ 0.050	$ 0.100
R6, R12	Vishay WSLP0805R0100FEA	$ 0.625	$ 1.250
R7, R8, R14, R15	*MF-RES-0805-470	$ 0.050	$ 0.200
Z1	NXP TDZ12J,115	$ 0.437	$ 0.437

The tabulated parts cost is $14.501, plus the cost of manufacturing the printed circuit board—quoted at $31.41 by the manufacturer—giving a subtotal of $45.911.

There was an additional $9.91 charge for use of parts that the manufacturer did not stock, and a labour cost of $18.05, bringing the total cost to $73.87. On the day of payment, this was equivalent to a cost of £56.33.

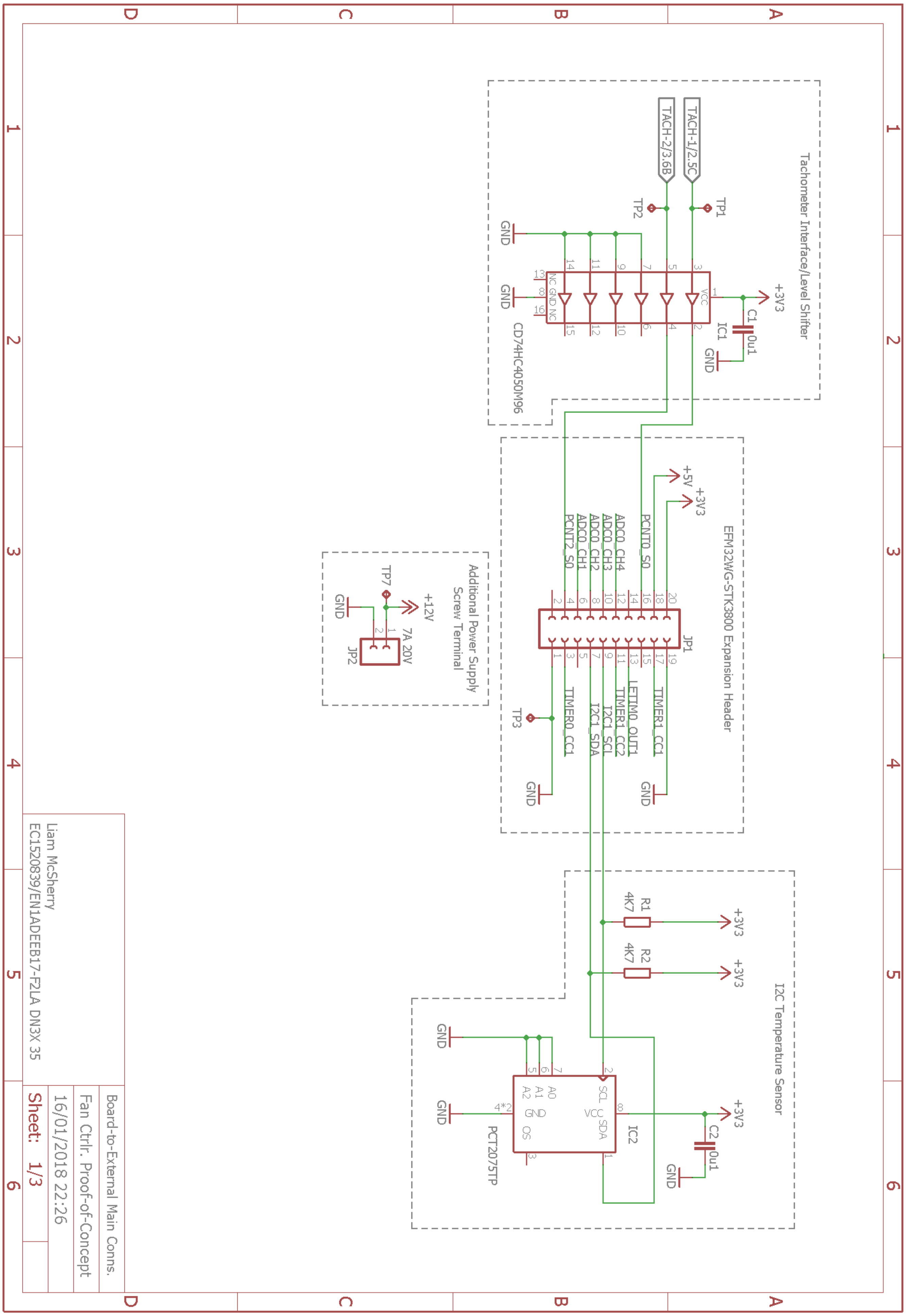
F2 Schematic Diagrams

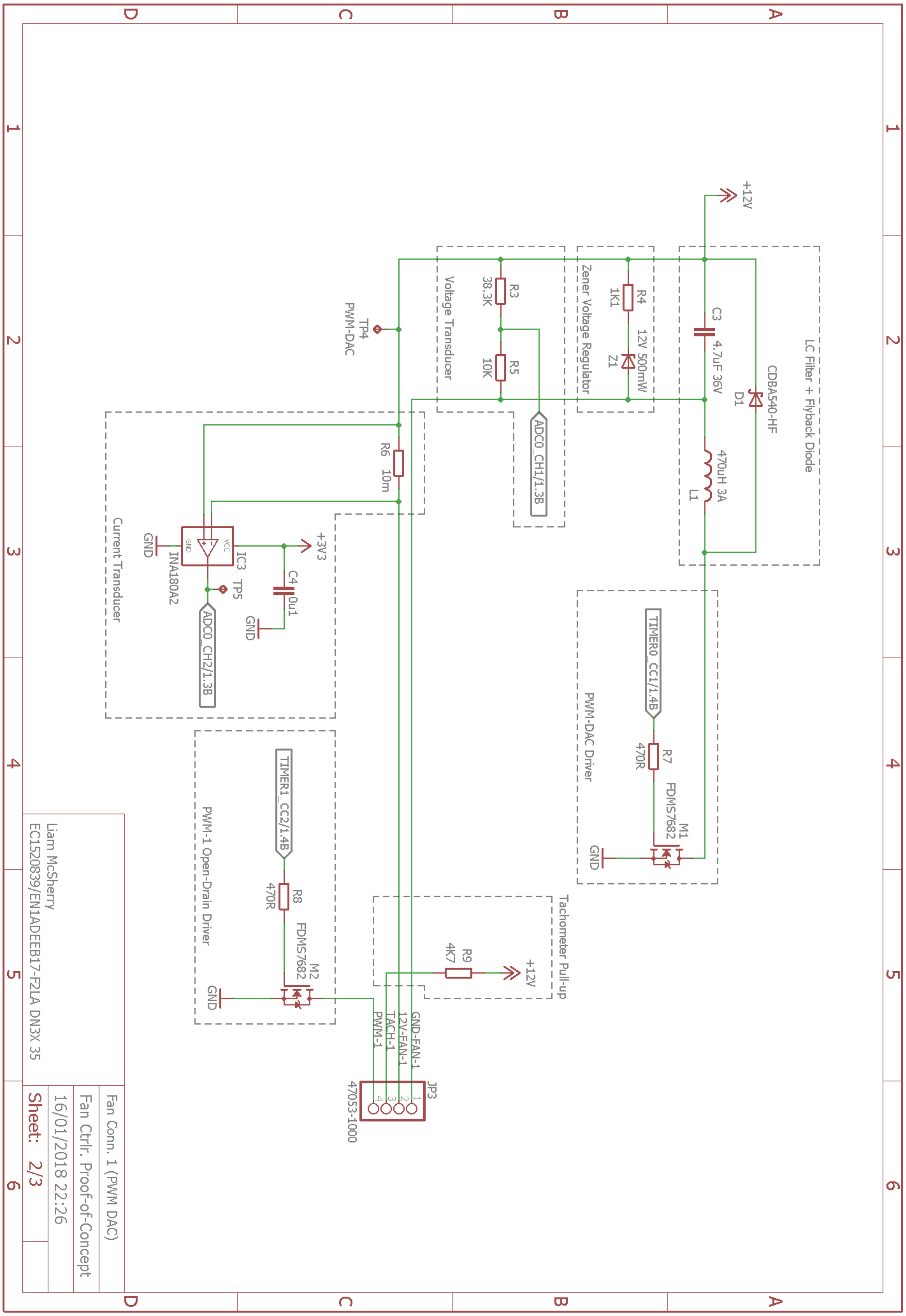
The schematic diagrams produced for the proof-of-concept prototype are given in the following pages. The diagrams were produced using Autodesk EAGLE.

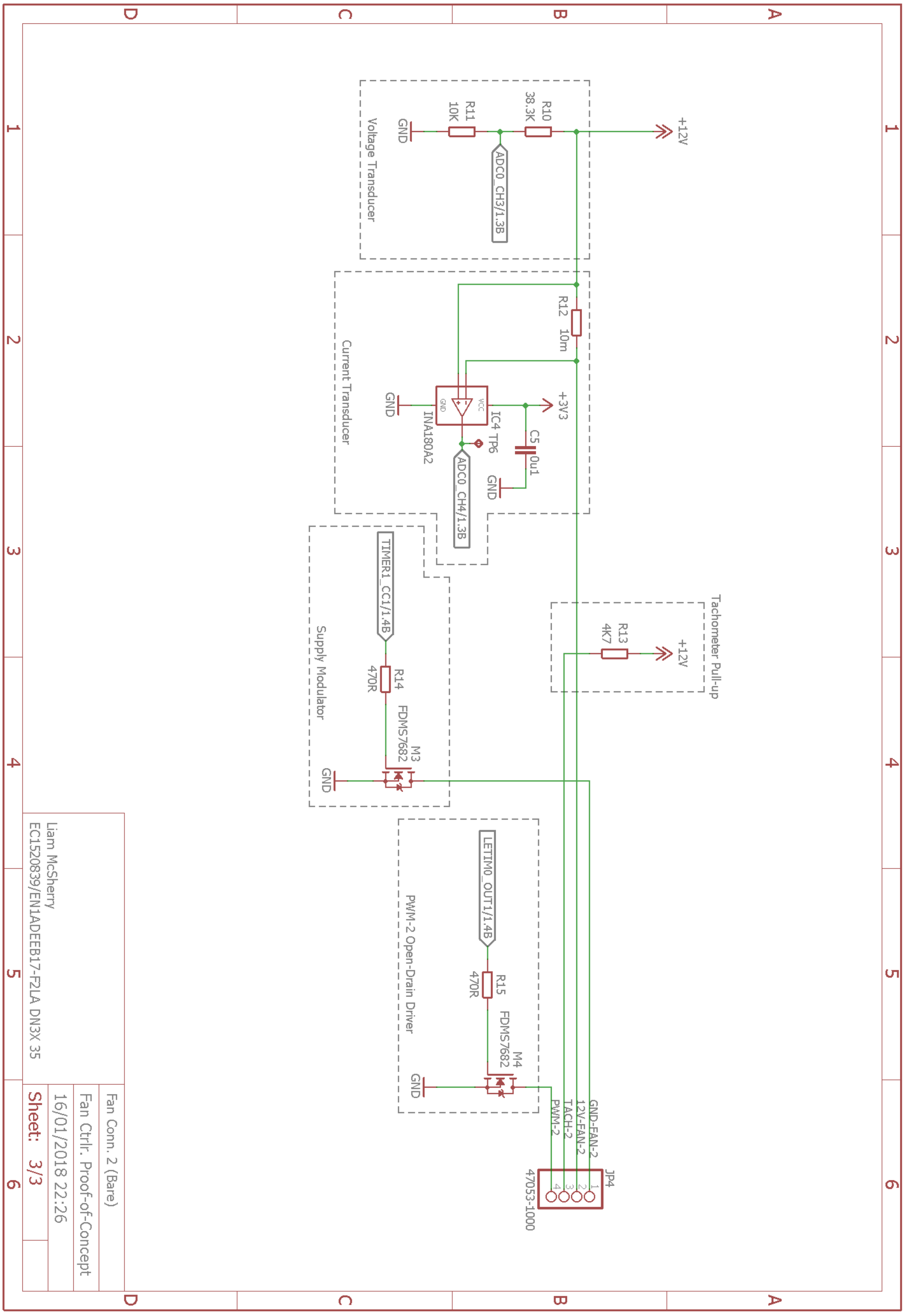
Sheet 1 of the diagrams (“board-to-external main conns.”) shows the connections from the proof-of-concept prototype circuit to external circuits, and relevant components which were considered to be most appropriately located in the same sheet as the external connections. The part marked JP1 is a 20-pin (10×2 rows) female header which mates with the 20-pin male header on the development kit identified as the expansion header (not to be confused with the debug connector). The part marked JP2 is a 2-pin screw terminal, to be used to connect a 12 V power supply to the prototype.

Sheet 2 of the diagrams (“fan conn. 1 (PWM DAC)”) shows the PWM DAC with its associated fan connection and the instrumentation to be used to monitor that fan connection.

Sheet 3 of the diagrams (“fan conn. 2 (bare)”) shows the fan connection with little more circuitry than is required to operate 4-pin varieties of fan. Inclusions other than that are the same instrumentation as is shown on sheet 2, and a further transistor for modulating the supply to the fan.


Tachometer Interface/Level Shifter
TACH-1/2.5C
TACH-2/3.6B
TP1
TP2
+3V3
C1
0u1
GND
IC1
CD74HC4050M96
EFM32WG-STK3800 Expansion Header
JP1
+5V
+3V3
PCNT0_S0
ADC0_CH4
ADC0_CH3
ADC0_CH2
ADC0_CH1
PCNT2_S0
TIMER1_CC1
LETIM0_OUT1
TIMER1_CC2
I2C1_SCL
I2C1_SDA
TIMER0_CC1
TP3
Additional Power Supply Screw Terminal
+12V
TP7
7A 20V
JP2
I2C Temperature Sensor
R1
4K7
R2
4K7
C2
0u1
IC2
PCT2075TP
SCL
SDA
VCC
GND
A0
A1
A2
OS
Liam McSherry
EC1520839/EN1ADEEB17-F2LA DN3X 35
Board-to-External Main Conns.
Fan Ctrlr. Proof-of-Concept
16/01/2018 22:26
Sheet: 1/3


+12V
LC Filter + Flyback Diode
C3
4.7uF 36V
CDBA540-HF
D1
470uH 3A
L1
Zener Voltage Regulator
R4
1K1
12V 500mW
Z1
Voltage Transducer
R3
38.3K
R5
10K
ADC0_CH1/1.3B
TP4
PWM-DAC
R6
10m
Current Transducer
IC3
INA180A2
VCC
GND
+3V3
C4
0u1
GND
TP5
ADC0_CH2/1.3B
PWM-DAC Driver
TIMER0_CC1/1.4B
R7
470R
M1
FDMS7682
GND
Tachometer Pull-up
R9
4K7
+12V
PWM-1 Open-Drain Driver
TIMER1_CC2/1.4B
R8
470R
M2
FDMS7682
GND
GND-FAN-1
12V-FAN-1
TACH-1
PWM-1
JP3
47053-1000
1
2
3
4
Liam McSherry
EC1520839/EN1ADEEB17-F2LA DN3X 35
Fan Conn. 1 (PWM DAC)
Fan Ctrlr. Proof-of-Concept
16/01/2018 22:26
Sheet: 2/3
A
B
C
D
1
2
3
4
5
6

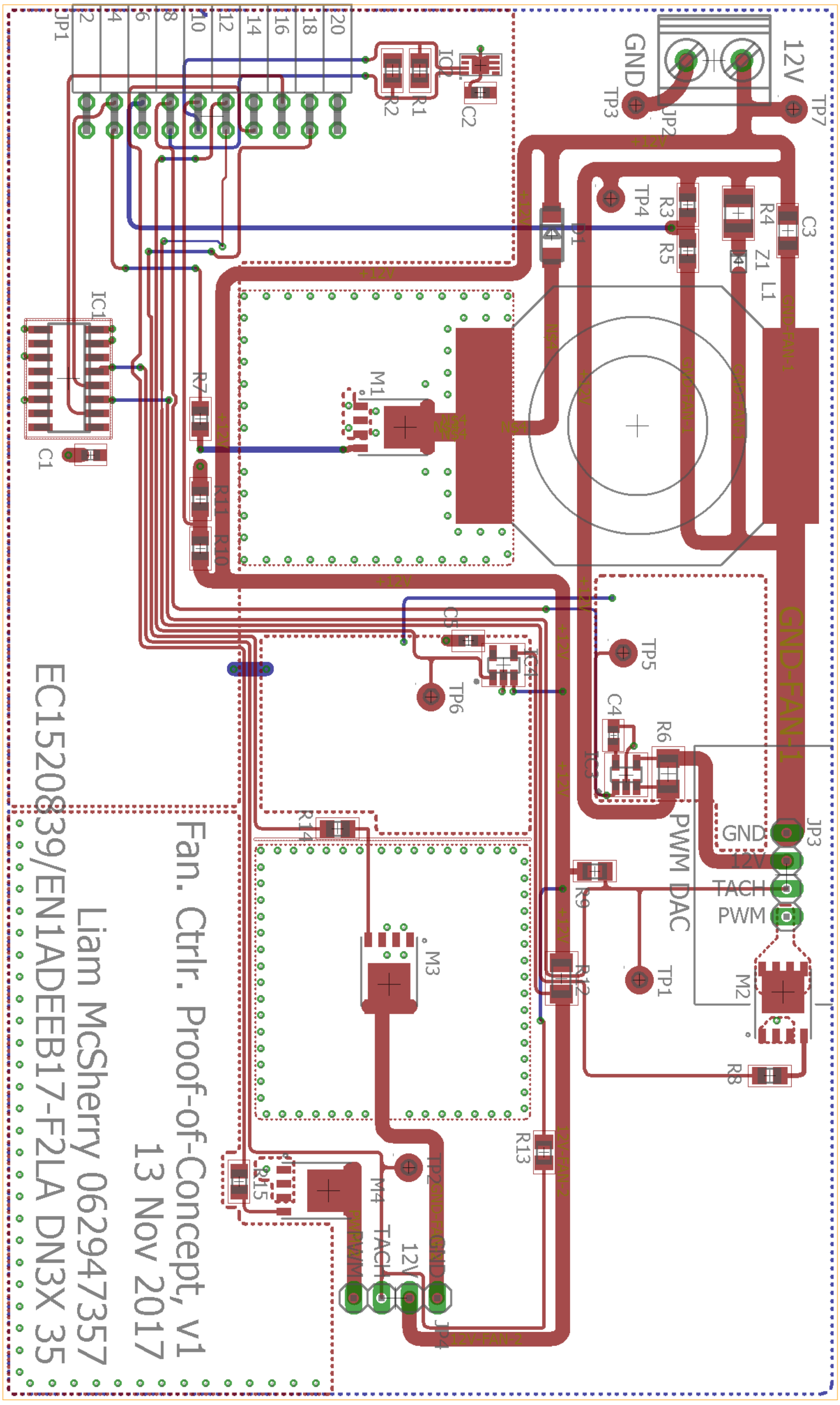

Voltage Transducer
+12V
R10
38.3K
R11
10K
GND
ADC0_CH3/1.3B
Current Transducer
R12
10m
+3V3
C5
0u1
GND
IC4
TP6
INA180A2
ADC0_CH4/1.3B
Tachometer Pull-up
R13
4K7
+12V
Supply Modulator
TIMER1_CC1/1.4B
R14
470R
M3
FDMS7682
GND
PWM-2 Open-Drain Driver
LETIM0_OUT1/1.4B
R15
470R
M4
FDMS7682
GND
GND-FAN-2
12V-FAN-2
TACH-2
PWM-2
JP4
47053-1000
Liam McSherry
EC1520839/EN1ADEEB17-F2LA DN3X 35
Fan Conn. 2 (Bare)
Fan Ctrlr. Proof-of-Concept
16/01/2018 22:26
Sheet: 3/3

F3 Circuit Designs

The circuit design produced for the proof-of-concept prototype is given on the following page. The design was produced using Autodesk EAGLE.

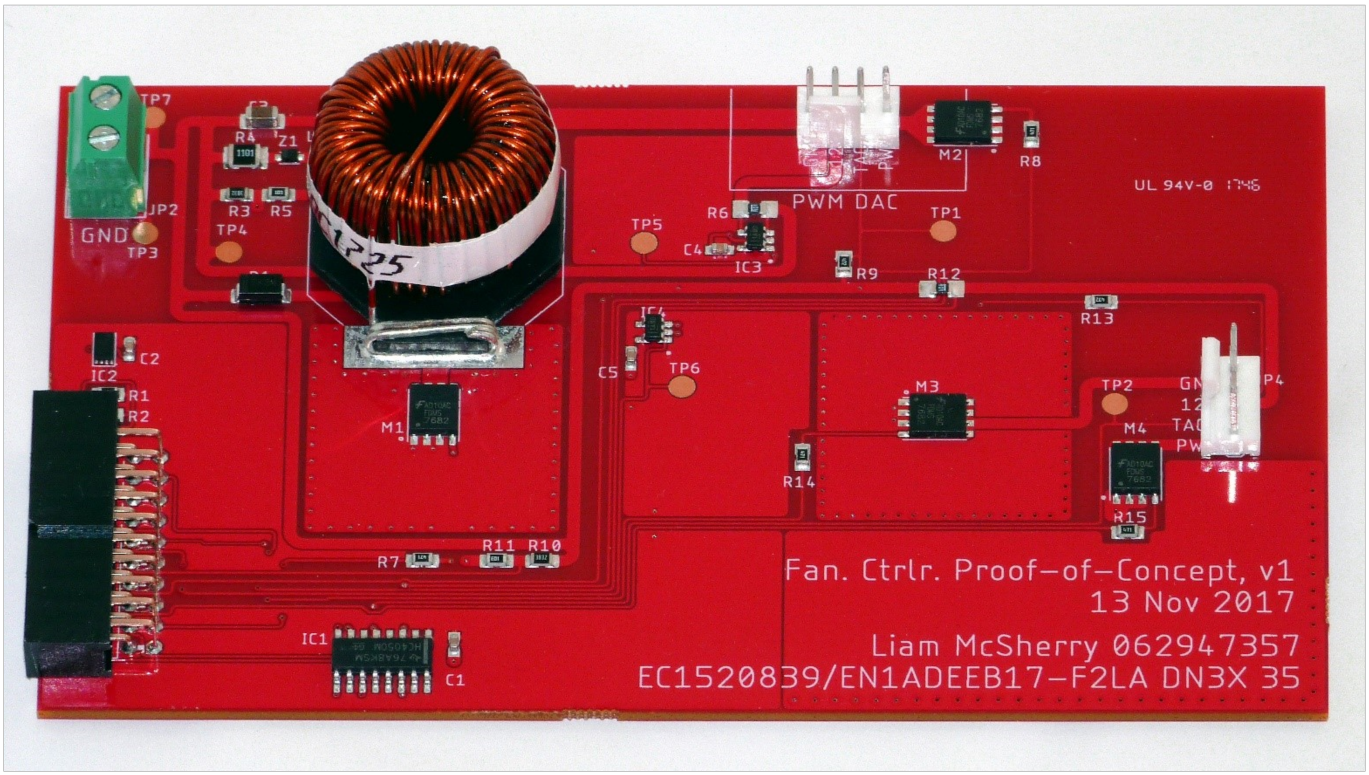
The design is for a two-layer circuit board, approximately 5×3 inches. The bottom layer is largely used as a ground plane and has minimal routing. The top layer is used for the majority of the routing, and has a mix of ground and +3.3 V supply planes, as well as bare areas without copper pours. In the design shown, dotted lines indicate the boundaries of copper pours.

Areas and dotted lines in red represent copper on the top layer. Areas and dotted lines in blue represent copper on the bottom layer. Green is used to represent a plated through-hole, with the green area being the copper pad. Grey is used for any documentation, and may or may not be present on the printed board.

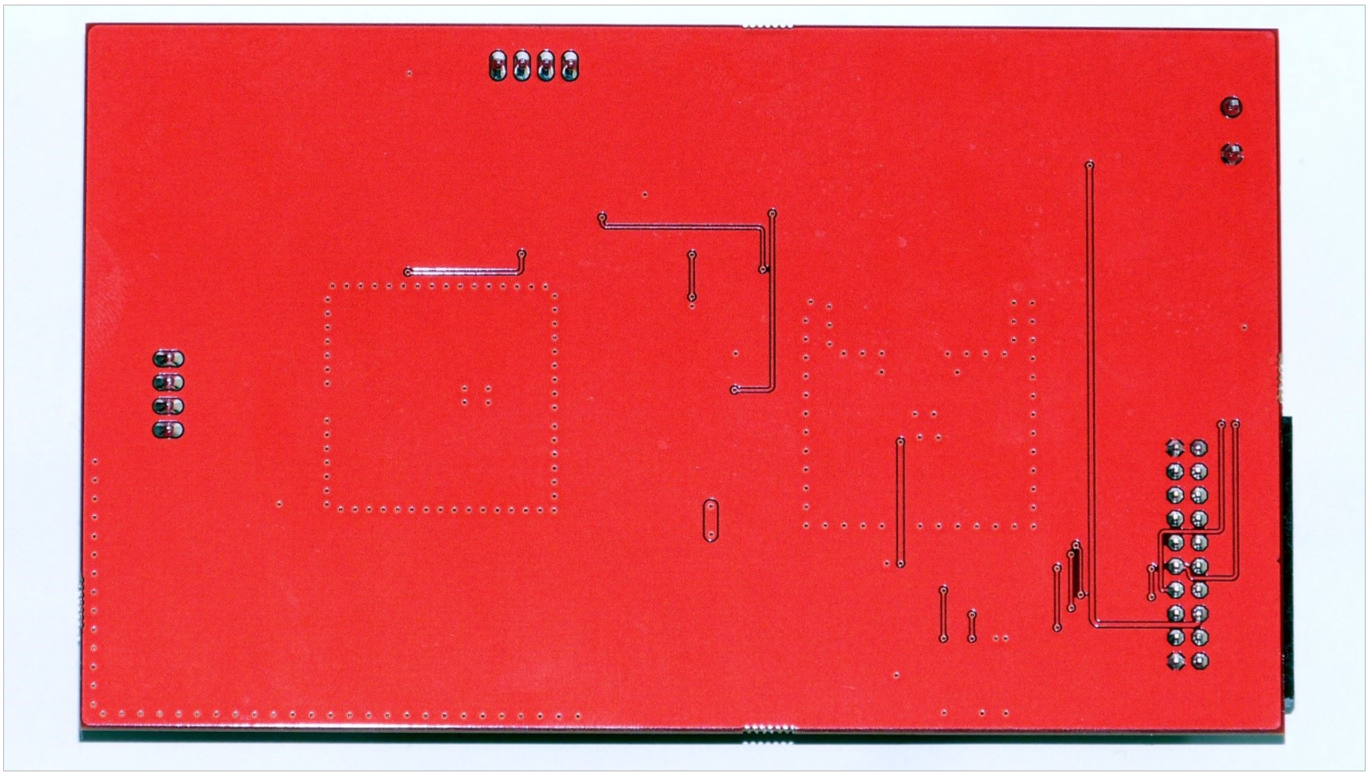


Fan. Ctrlr. Proof-of-Concept, v1
13 Nov 2017
Liam McSherry 062947357
EC1520839/EN1ADEEB17-F2LA DN3X 35
GND
12V
TP3
TP7
JP2
JP1
2
4
6
8
10
12
14
16
18
20
IC2
R1
R2
C2
TP4
R3
R4
C3
D1
R5
Z1
L1
IC1
C1
R7
R11
R10
M1
C5
IC4
TP5
TP6
C4
R6
IC3
PWM DAC
JP3
GND
12V
TACH
PWM
R14
R9
M3
TP1
M2
R12
R8
R13
TP2
M4
R15
12V
TACH
PWM
GND
JP4
+12V
GND-FAN-1
12V-FAN-2

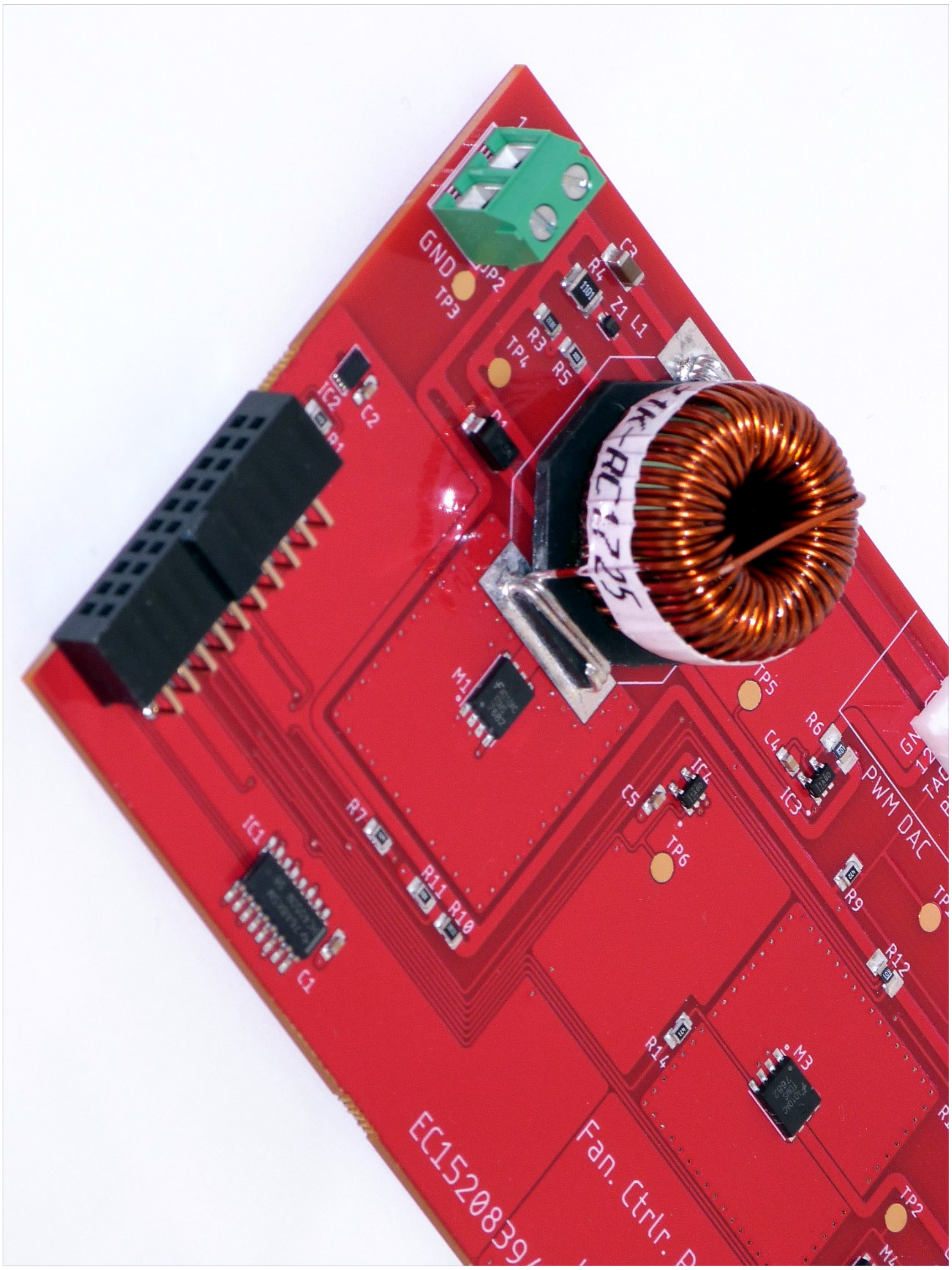
F4 Photographs



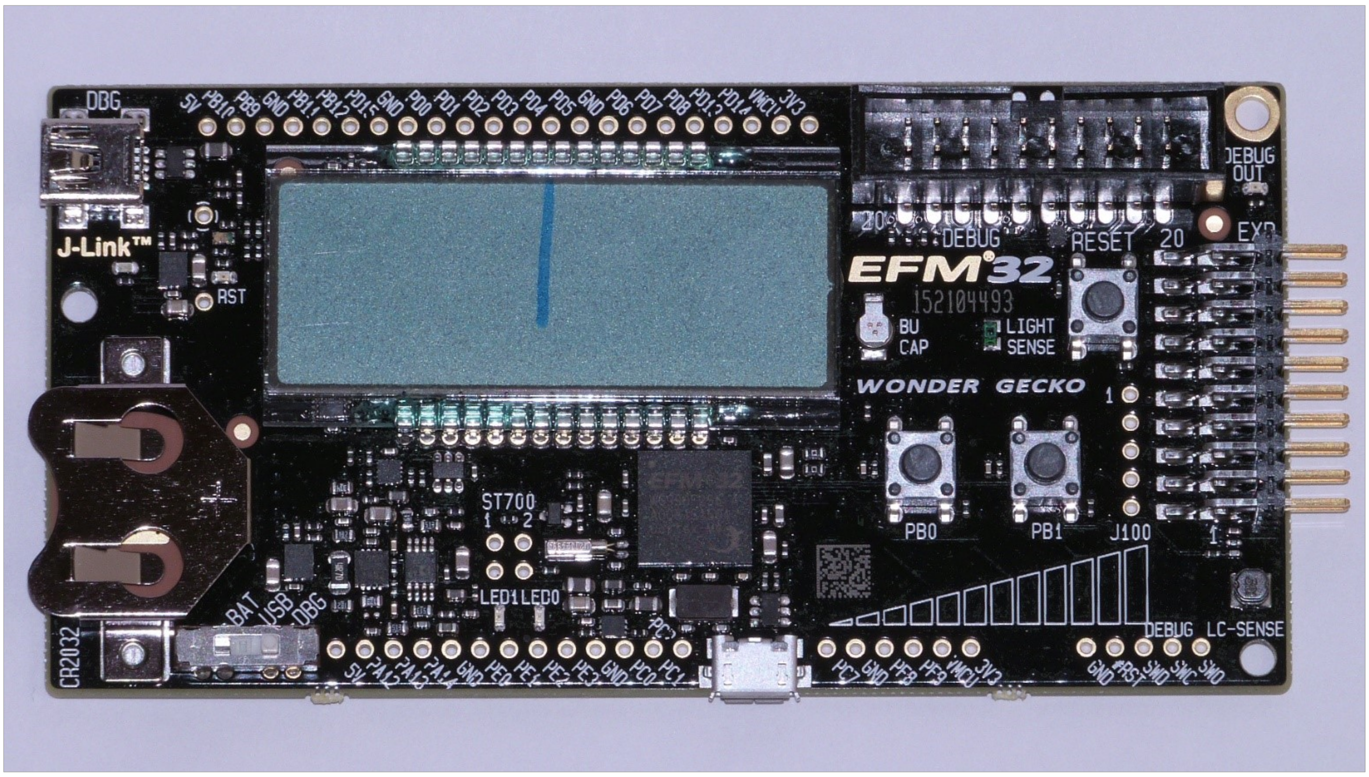
Face, front view of the proof-of-concept prototype.

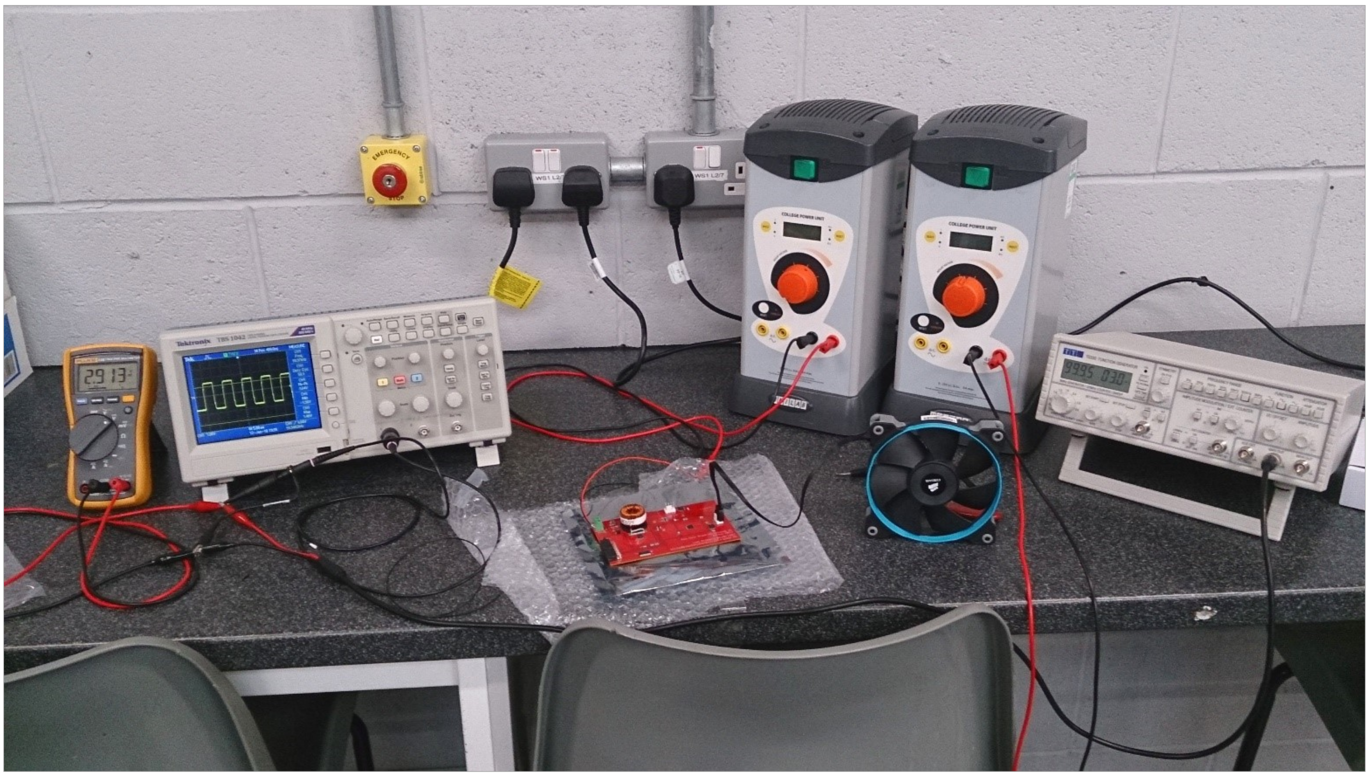


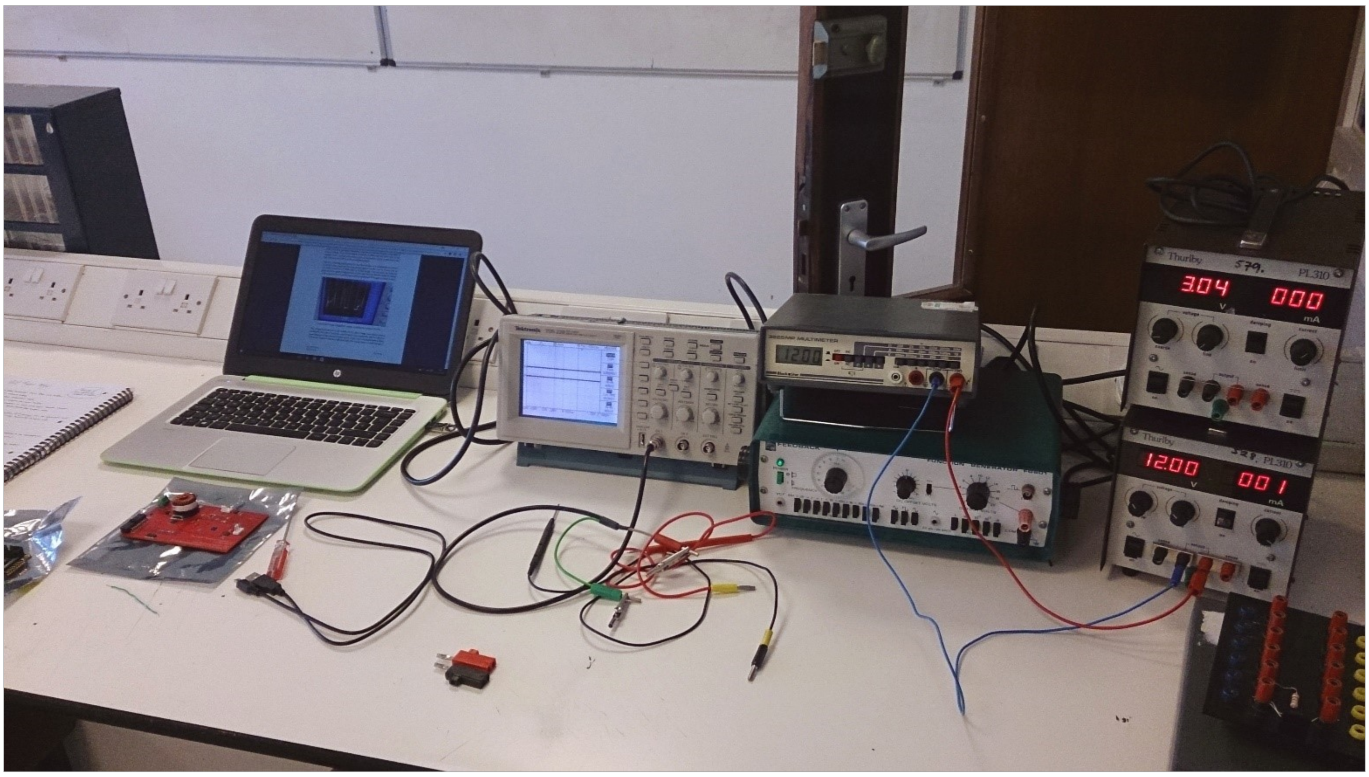
Reverse, front view of the proof-of-concept prototype.



Face, side view of the proof-of-concept prototype, showing the
PWM DAC, screw terminal, development kit connector, and level shifter.


The development kit, with USB debugger (top left), LCD (top centre), ARM Cortex-M controller (centre), USB connector (bottom centre), and expansion header (right).


The test setup for the proof-of-concept prototype on the 12th of January 2018.



The test setup for the proof-of-concept prototype on the 1st of February 2018.

F5 Firmware Design

F5.1 Initial flowcharts

The initial flowcharts produced to give a high-level outline of the firmware are given on the following pages. The flowcharts were produced using Dia.

Summary: first page

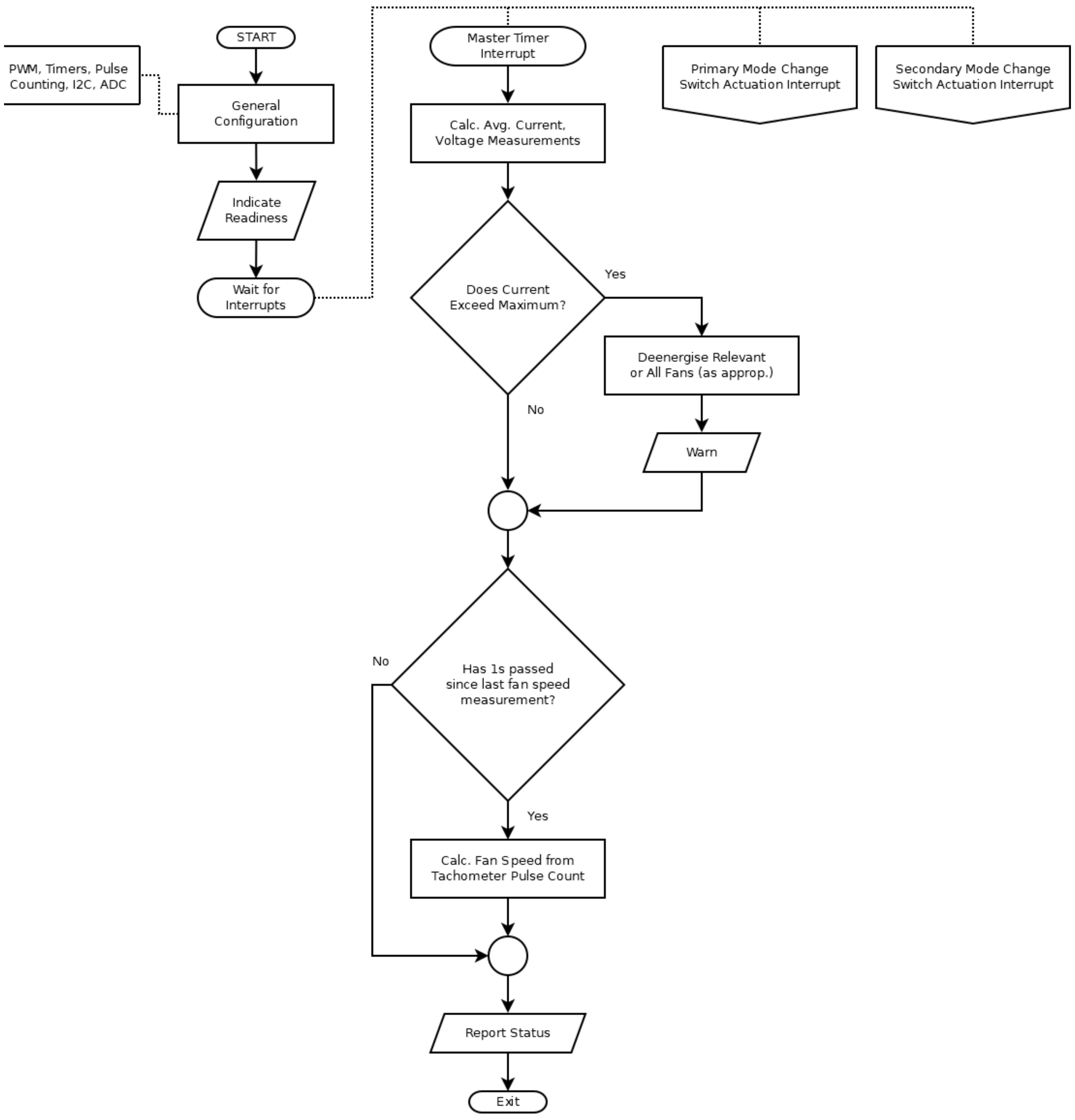
The first page shows the general design of the entry point and the master timer interrupt service routine portions of the firmware. The entry point performs the initial setup of the microcontroller, the general configuration for all required hardware peripherals (such as pulse-width modulation generators, timers, pulse counters, I^2C communication, and analogue-to-digital converters) and enables the master timer interrupt before halting and waiting for interrupts.

The master timer is a periodic timer used to periodically make measurements. It is expected that the timer will be set to a 10 ms interval—sufficiently fast to continuously average voltage and current values, the intention of which is to reduce the effects of voltage transients in the lines connecting the transducer to the microcontroller.

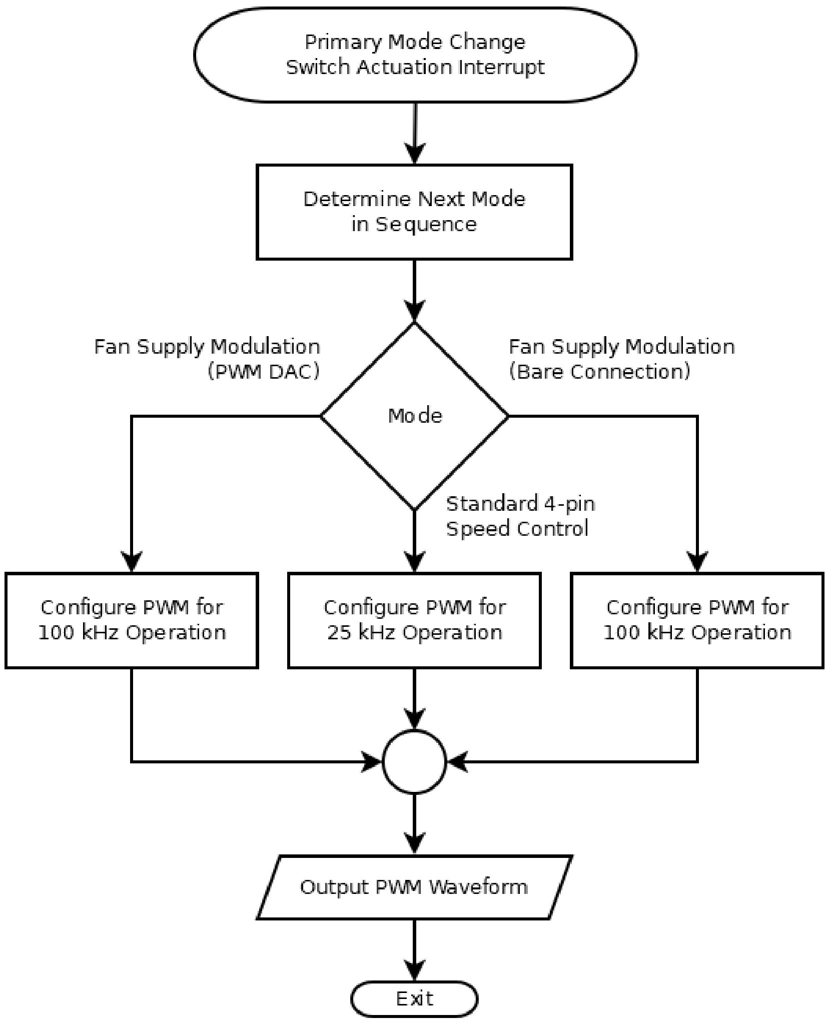
Additionally, the master timer would be used to time the one-second interval between reading the number of tachometer pulses counted—likely implemented through a software counter (counting through 1000 ms / 10 ms = 100 values) that is incremented on each master timer interrupt, the one-second interval allows a sufficient number of pulses to be recorded for relatively accurate measurement of fan speed. In one second, a 500 rpm fan would complete 8⅓ revolutions, which would result in approximately 16 or 17 pulses being recorded (given that there are two pulses produced per revolution). Multiplying this pulse count by $60\times\frac{1}{2}$ to approximate a per-minute number of revolutions gives 480, which is not hugely different from the 500 rpm nominal speed.

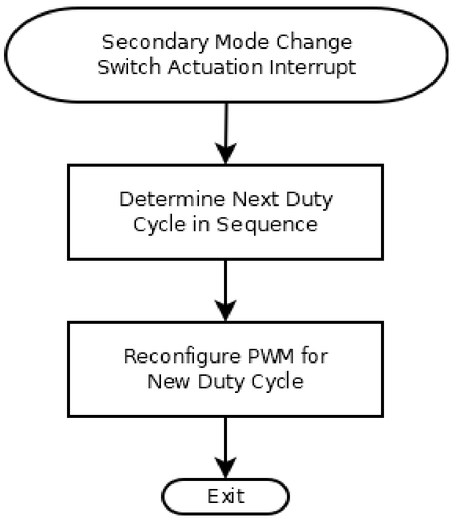
Summary: second page

The second page shows the "primary" and "secondary mode change switch actuation interrupts," which are the interrupts produced when a primary and a secondary switch are actuated. The primary switch is used to rotate between the available modes, and the secondary switch between duty cycles in each mode. A switch between modes would largely consist of reconfiguring the microcontroller PWM peripherals to disable whichever peripheral was outputting on actuation and enable whichever is required to output for the next mode in sequence.



START
PWM, Timers, Pulse Counting, I2C, ADC
General Configuration
Indicate Readiness
Wait for Interrupts
Master Timer Interrupt
Primary Mode Change Switch Actuation Interrupt
Secondary Mode Change Switch Actuation Interrupt
Calc. Avg. Current, Voltage Measurements
Does Current Exceed Maximum?
Yes
Deenergise Relevant or All Fans (as approp.)
Warn
No
No
Has 1s passed since last fan speed measurement?
Yes
Calc. Fan Speed from Tachometer Pulse Count
Report Status
Exit


Primary Mode Change Switch Actuation Interrupt
Determine Next Mode in Sequence
Mode
Fan Supply Modulation (PWM DAC)
Fan Supply Modulation (Bare Connection)
Standard 4-pin Speed Control
Configure PWM for 100 kHz Operation
Configure PWM for 25 kHz Operation
Configure PWM for 100 kHz Operation
Output PWM Waveform
Exit


Secondary Mode Change Switch Actuation Interrupt
Determine Next Duty Cycle in Sequence
Reconfigure PWM for New Duty Cycle
Exit

F5.2 Specific considerations: fan speed measurement

The microcontroller on the development kit has three pulse counters—1×16-bit and 2×8-bit. In order to ensure that fan speed measurement through the counters works sensibly and consistently, any decisions taken must consider only the capabilities of the 8-bit counters, and not those of the 16-bit counter.

Measurable fan speeds

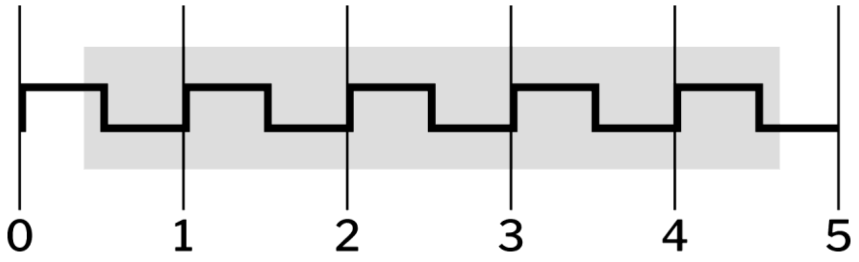
For the simplest solution—a periodic check of a counter's value—the greatest impact on measuring ability is from the choice of counting period. This period varies upper limit, depending also on the maximum counter value, of measurable fan speeds. For example, if the counter is an 8-bit counter, with a counting period T, and considering that a fan produces two pulses per revolution, the maximum fan speed is $(2^8-1)/2$ revs per T seconds. If T is 1, the greatest fan speed which the system can measure is 7650 rpm.

This maximum measurable speed can be increased by shortening the counting period, but doing so impacts the minimum measurable fan speed. The speed of a motor may only be an average speed over time, and so the motor may operate faster than nominal at one time and slower than nominal at another. The shorter the counting period, the more likely it is that the period will cover either but not both of these times, and so the more likely it is that large variations in fan speed will be recorded. Lengthening the counting period reduces this likelihood at the cost of decreasing the maximum measurable fan speed. In this simplest solution, it would be required to select a counting period which was an appropriate trade-off between maximum and minimum measurable speeds.

However, the pulse counters on the microcontroller include features which make a more complex system without this trade-off practical. The counters are capable of generating an interrupt when the maximum is reached, and so by registering and servicing this interrupt firmware on the microcontroller could count past the maximum. This would enable the use of a counting period long enough to give a useful result at low speeds without affecting the measurable maximum in a way that is significant. To clarify, any further limit would be the result of the storage of the firmware-based counter—for example, if this count were stored in a 16-bit variable, it would be "limited" to $(2^{16}-1)/2$ revs per T seconds. If T is 1, then this places the limit at nearly 2 million rpm.

Low-speed measurement accuracy

Depending on the operation of the pulse counter, it is not inconceivable that an additional pulse could be registered. This is illustrated in the below diagram.



If the counting period is the shaded area, it can be seen that there are four full pulses, but five periods where the signal is high. If the pulse counter were level-sensitive, the partially-shaded pulse might register as two pulses. At particularly low fan speeds, this could greatly affect the accuracy of the measured fan speed.

For example, if a 600 rpm fan were operated at 180 rpm (or 30% full speed[1]), it would nominally produce 2×180 = 360 pulses per minute. However, the speed of a fan is permitted to vary by ±10% of full speed from the speed specified by the control signal, and so the fan could instead be operating at 120 rpm and producing pulses at a rate of 240 per minute. In a counting period of one second, the fan would then nominally produce four pulses. However, if a fifth is registered, the fan would appear to be operating at 150 rpm instead of 120 rpm—an error of 25% the true speed. At full fan speed, a single additional pulse has less of an effect—a further pulse would continue to increase the measurement by 30 rpm, but this might only represent 5% of the 600 rpm true speed.

The pulse counters on the microcontrollers provide some features which negate these issues. The counters can be configured to trigger on either the rising or the falling edge, and can be configured to require a certain "hold time" (that is, a time for the duration of which the signal must remain in the high state to be registered) to register a pulse. By configuring the counters to trigger on the positive edge, the issue of the counting period starting in the middle of a pulse (as illustrated above) is eliminated. Through the use of the hold time requirement—specifically, a requirement for a pulse to last for five clock cycles, equivalent to around 0.15 ms when the counters are driven from the 32.768 kHz clock—pulses generated by interference can be partially, if not almost entirely, eliminated.

If an improvement in accuracy is necessary, it would be possible to use a rolling average. While on first thought this may not appear practical—over a number of seconds, for example, a fan could change speed many times, and so it could be thought that the accuracy of the average would degrade with time—the controller determines when the fan speed is to change, and would be able to reset the rolling average as required. It remains possible for external influences, such as the blades of the fan becoming obstructed, to change the fan speed, but such events could be detected by monitoring for sustained and significant changes in the raw fan speed. This would be somewhat analogous to integral–derivative control.

Basic testing and verification

At first consideration, it would appear that the only possible means of testing fan speed measurement would be to connect the development kit to appropriate hardware—either to the proof-of-concept prototype and a fan, or to a signal generator emulating the tachometer output of a fan. This would be not at all preferable, as it would prevent testing of the firmware for fan speed measurement otherwise than during the limited laboratory time.

However, this is not the case. The microcontroller incorporates what is branded a "peripheral reflex system" (PRS), which connects hardware devices included in the microcontroller ("peripherals") and allows the communication of events from one peripheral to another, enabling the consumer of the event to respond without intervention from the central processor (hence, "reflex"). Through this system, a pulse counter can operate as a consumer and so can be configured to count pulses

[1] That speed being the greatest permissible value for a PWM-controlled fan's minimum controlled speed. That is, where the speed of the fan is specified by a PWM control signal, the greatest value below which the fan need not respond.

generated by other peripherals. In combination with a timer configured to signal on the PRS when it fires, a periodic tachometer pulse can be emulated internally within the microcontroller without the requirement for a laboratory or special equipment and whilst still making use of the pulse counter peripheral (rather than an abstraction over the peripheral where the behaviour is emulated in firmware).

If testing the pulse counters using the PRS, the counters must be configured to operate in single external input oversampling mode. In this mode, the counters are clocked by their input, rather than the LFA clock. When the counters were instead configured to operate in single-input oversampling mode, which is synchronised to the LFA clock, pulses were not reliably counted. This is likely an artefact of the PRS, as pulses on the PRS are one clock cycle long. If not generated exactly in time with a pulse from LFA, the single pulse would never be registered. This is not anticipated to be an issue for a connection to a real fan, as tachometer pulses will be significantly longer.

F5.3 Specific considerations: microcontroller PWM generators

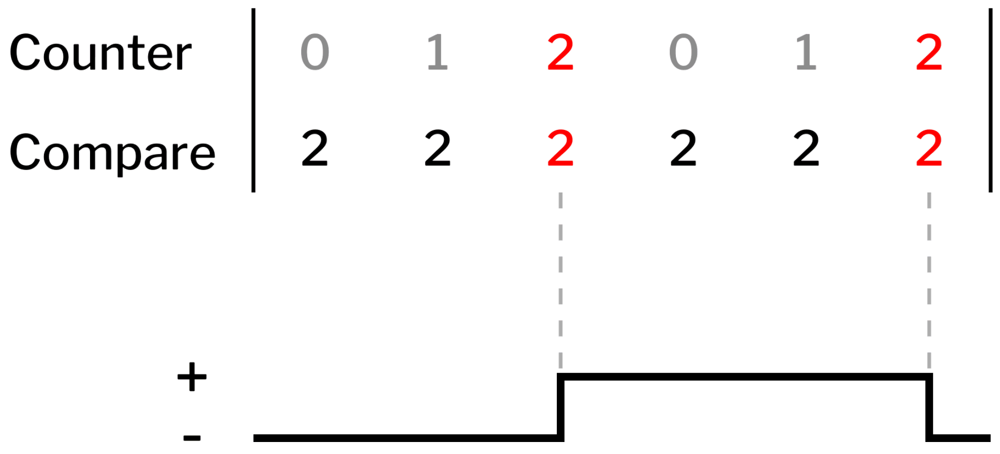
As shown in the schematic diagrams, the proof-of-concept prototype uses both the microcontroller's normal timers and the microcontroller's low-energy timer to produce the PWM waveforms required. As the configuration for normal and low-energy timers is different, and as the use of these timers to produce PWM is not entirely straightforward, this appendix provides additional explanation.

The information provided in this section is available in the reference manual for the microcontroller, but the manual is not considered especially clear.

Anatomy of the timers

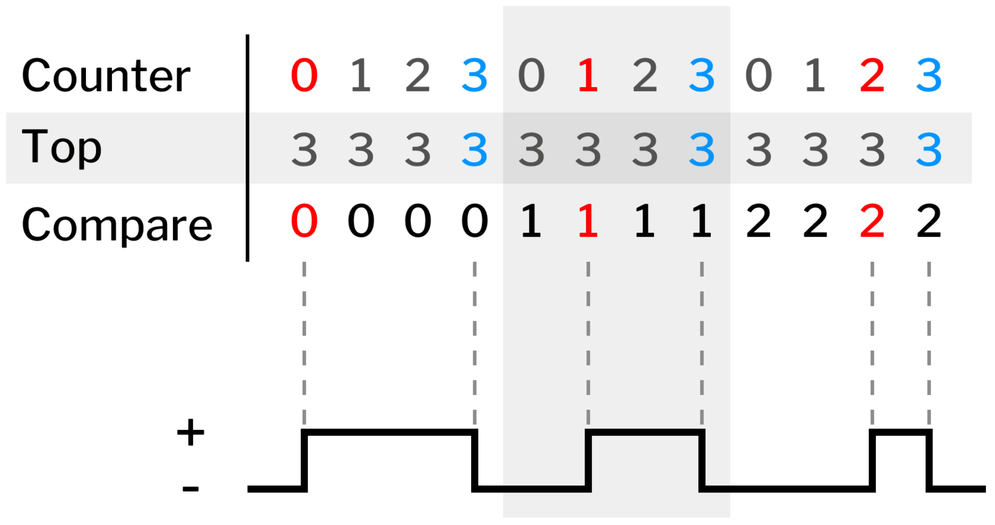
For the purpose of generating a PWM waveform, the normal and low-energy (LE) timers are conceptually identical—each consists of a counter capable of counting in either direction (up or down) and interrupting the processor when the count reaches zero or a setpoint, and each has an associated set of output channels able to process and act on the count to produce an output signal (which will generally be output from the microcontroller on one of its pins).

The processing available to the output channels is basic, but important for this use is that each of the timers supports a toggle-on-match function. That is, when the count equals a channel-specific value, the channel will toggle the output of the channel (so that high becomes low and low becomes high). This function is illustrated below. For simplicity, the compare value is also the value at which the counter overflows to zero.



As it stands above, this function is not particularly useful for producing a PWM waveform—varying the compare value varies the width of the pulse, but this can only ever produce fixed-width pulses and so could only ever produce a waveform with a duty cycle of 50% (or a mark–space ratio of 1:1).

However, with a minor modification, this function can be useful in producing PWM waveforms. If the output channel, each time the counter overflows (in the case of an up-counter) or underflows (for a down-counter), is reset to a fixed idle state, then the compare value can be used to control the period of time the signal is in the opposite state. Both the normal and LE timers support this method and, while other methods exist, this method is likely to be one of the simpler options available (in terms of code complexity) for producing PWM signals. This modified function is illustrated by the below diagram, and the value labelled “top” is the value at which the counter overflows to zero.



As can be seen, this minor modification enables the production of a variable-duty-cycle waveform, an essential requirement for a PWM generator. By also varying the “top” value for the counter, both the frequency of the wave and the resolution of duty cycle steps are varied.

At a high level, both the normal and LE timers on the microcontroller operate as illustrated in the above diagram. More practically, there are not insignificant differences in operation which require different treatment in any code produced.

Normal timers

In the case of the normal timers, there is a high degree of flexibility. Each timer has three output channels,[2] each of which can act when the counter over- or underflows, or when the counter matches a channel-specific value.

If the timer were to be configured as an up-counter which toggled the input when the counter was equal to a given value, the output waveform would remain high until the register `TIMERn_CNT` became equal to `TIMERn_CCx_CCV`, at which point it would transition to low. The waveform would then transition back to the high state when `TIMERn_CNT` became equal to `TIMERn_TOP` and overflowed to zero.

In this case, the PWM frequency is given by the same equation as the timer frequency would normally otherwise be given—that is, the frequency is equal to the timer clock divided by the product of the prescaler setting and one more than the counter's top value $f_{PWM} = f_{HFPER}/[(PRESCALER×(TOP+1))$. To attain 100 kHz, as required for the PWM DAC input, a prescaler value of 1 and a TOP of 139 is required. While it is possible to attain this frequency with other values, this pair of values has the advantage that, to attain the 25 kHz control signal used with standard 4-pin fans, the prescaler can be adjusted from 1 to 4. Other values (such as a prescaler of 4 and a TOP of 34) would produce the same effect, but would require that the prescaler be adjusted from 4 to 16, which leaves room for error. For example, it is not unforeseeable that code written absentmindedly would adjust the prescaler from 4 to 8 (four more) instead of 4 to 16 (four times more).

Further, as the TOP value determines the resolution of the PWM duty cycle, the use of a value of 139 provides a resolution of approximately 0.7%, which is far finer than the approximately 3% provided by a TOP of 34. This enables a user to set fan speed (where this is determined by PWM) in near-1% increments, which makes more intuitive sense than adjustment in 3% increments.

Low-energy timers

The LE timers are relatively inflexible. While the LE timers still have two output channels, and while each of these channels can produce a signal different from that of the other channel, both channels are driven from a single comparator (in comparison to the normal timers, which have a comparator for each channel).

Further, the cues for acting and the action taken on those cues is fixed. The timer begins counting down from the value in the register `COMP0`. When the count matches `COMP1` and if the repeat counters[3] `REP0` and `REP1` are non-zero, the signal becomes high. When the counter underflows (irrespective of the value of the repeat counters), the signal is reset to a configurable idle value.

The LE timer, being a low-energy peripheral, would normally be clocked by the Low-Frequency RC Oscillator (LFRCO) through the LFA clock. However, as the LFRCO operates at 32.768 kHz, this is not possible—the 21–28 kHz accepted by a standard 4-pin fan as a PWM control signal cannot be produced using dividers

[2] In reality, the "output channels" for the normal timers are actually "capture/compare channels," and provide functionality both for generating output from and processing input to the timer. The precise input functionality is outside the scope of this note.

[3] The repeat counters `REP0` and `REP1` are used in other timer modes to, for example, produce a pulse or signal a specified number of times.

and prescalers present on the microcontroller. Instead, the LFA clock must use the 7 MHz $\text{HFCORECLK}_{\text{LE}}$.[4] In that configuration, the required 25 kHz can be produced by configuring the LE timer's prescaler to divide the clock by 2 and by setting `COMP0` to 139 (i.e. having the timer count down from 140). As the LE timer is only used to provide a PWM control signal to the fan (and not to modulate the supply or control the PWM DAC), it is not necessary to configure the timer to produce a PWM signal with a frequency of 100 kHz.

[4] Note that, while $\text{HFCORECLK}_{\text{LE}}$ is specified here as 7 MHz, its frequency is derived from the HFCORE clock which is generally derived from the HFRCO. Its frequency is half or one quarter that of the clock from which it is derived.

F6 Test Plan

F6.1 General considerations

The overall objective in producing the proof-of-concept prototype circuit is both to enable the testing the firmware and software elements, and to ensure that the methods of control selected function safely and as intended.

As it is uncertain whether the PWM DAC-based control circuit will function as desired, and considering that there is the potential for the mis-operation of a fast-switching inductor to produce large voltages, any test involving the PWM DAC and the development kit should be performed after other tests. This will ensure that, if the PWM DAC causes damage to equipment, some usable data will have been collected. In addition, where any test involves the PWM DAC circuit but not the development kit, it would be advisable to use the Corsair Air Series SP120 PWM fan—two of this type of fan were purchased, and so the loss of one of those fans would not reduce the variety of fans available for testing.

F6.2 Equipment listing

The following equipment is to be used in the testing of the proof-of-concept prototype circuit. Supplies and instrumentation are not listed.

Designator	Description
Dev. Kit	Silicon Labs EFM32WG-STK3800 development kit.
—	Proof-of-concept prototype circuit.
Fan 1A Fan 1B	Corsair Air Series SP120 PWM fan.
Fan 2	Fractal Design Silent Series R3 fan.
JP1	Proof-of-concept prototype circuit, component JP1. EFM32WG-STK3800 expansion header.
JP2	Proof-of-concept prototype circuit, component JP2. Additional power supply screw terminal.
JP3	Proof-of-concept prototype circuit, component JP3. PWM DAC fan connector.
JP4	Proof-of-concept prototype circuit, component JP4. Bare fan connector.
TP1	Proof-of-concept prototype circuit, test point 1. Tachometer 1, pre-level shift.
TP2	Proof-of-concept prototype circuit, test point 2. Tachometer 2, pre-level shift.
TP4	Proof-of-concept prototype circuit, test point 4. PWM DAC output voltage.
TP5	Proof-of-concept prototype circuit, test point 5. PWM DAC fan connection, current transducer output.

Designator	Description
TP6	Proof-of-concept prototype circuit, test point 6. Bare fan connection, current transducer output.
TP7	Proof-of-concept prototype circuit, test point 7. Additional power supply, 12 V line.
USB 1	Silicon Labs EFM32WG-STK3800, interface USB port. Board controller interface USB Mini-B (left edge).
USB 2	Silicon Labs EFM32WG-STK3800, EFM32 USB port. Microcontroller USB Micro-AB (bottom edge).

Where a voltage is given, that voltage is referenced to ground unless otherwise specified.

F6.3 Action items

1 Basic verification: bare fan connection

A. Connect a 12 V supply to JP2.
B. Connect Fan 1A to JP4.
C. Connect a 3.3 V supply to JP1 pins 1 and 19 (GND), 17 and 20 (+3V3).
D. Observe that Fan 1A starts and continues to rotate.
E. Observe at TP6 that the voltage does not exceed approximately 0.1 V after allowing at least 1 second to expire from the completion of step C.
F. Observe at TP2 that the voltage does not exceed 12.6 V.
G. Observe at JP1 pin 10 that the voltage does not exceed approx. 2.6 V.

2 Basic verification: PWM DAC fan connection

A. Connect a 12 V supply to JP2.
B. Connect Fan 1A to JP3.
C. Connect a 3.3 V supply to JP1 pins 1 and 19 (GND), 3 and 20 (+3V3).
D. Observe that Fan 1A starts and continues to rotate.
E. Observe at TP5 that the voltage does not exceed approximately 0.1 V after allowing at least 1 second to expire from the completion of step C.
F. Observe at TP1, TP4, and TP 7 that the voltage does not exceed 12.6 V.
G. Observe at JP1 pin 6 that the voltage does not exceed approx. 2.6 V.

3 Basic verification: MCU PWM DAC control mode

A. Set the Dev. Kit power source select switch to "DBG."
B. Connect through USB 1 the Dev. Kit to a computer.
C. Configure the Dev. Kit to operate in the first mode in the program specification (see section 14.4.1).
D. Operate the Dev. Kit in this first mode with all duty cycle variants and make the following observations.
E. Observe at the Dev. Kit expansion header pin 3 (PC0 / TIMER0_CC1) that a PWM waveform, approx. 100 kHz, correct duty cycle is present.
F. Observe at the Dev. Kit expansion header pin 11 (PB11 / LETIM0_OUT1) that no voltage is present.

4 Basic verification: MCU modulated supply control mode

A. Set the Dev. Kit power source select switch to “DBG.”
B. Connect through USB 1 the Dev. Kit to a computer.
C. Configure the Dev. Kit to operate in the second mode in the program specification (see section 14.4.2).
D. Operate the Dev. Kit in this second mode with all duty cycle variants and make the following observations.
E. Observe at the Dev. Kit expansion header pin 17 (PD7 / TIMER1_CC1) that a PWM waveform, approx. 100 kHz, correct duty cycle is present.
F. Observe at the Dev. Kit expansion header pin 13 (PB12 / TIMER1_CC2) that no voltage is present.

5 Basic verification: MCU PWM control signal control mode

A. Set the Dev. Kit power source select switch to “DBG.”
B. Connect through USB 1 the Dev. Kit to a computer.
C. Configure the Dev. Kit to operate in the third mode in the program specification (see section 14.4.3).
D. Operate the Dev. Kit in this third mode with all duty cycle variants and make the following observations.
E. Observe at the Dev. Kit expansion header pin 13 (PB12 / TIMER1_CC2) that a PWM waveform, 21–28 kHz, correct duty cycle is present.
F. Observe at the Dev. Kit expansion header pin 17 (PD7 / TIMER1_CC1) that a constant voltage, approx. 3.3 V, is present.

6 Basic verification: MCU pulse counting

A. Set the Dev. Kit power source select switch to “DBG.”
B. Connect through USB 1 the Dev. Kit to a computer.
C. Connect a signal generator to the Dev. Kit expansion header pins 1 and 19 (GND) and 4 (PD0 / PCNT2_S0 / +3V3).
D. Configure the Dev. Kit to operate in the third mode in the program specification (see section 14.4.3).
E. Configure the signal generator to output a square wave of frequency approximately 78.33 Hz, duty cycle 50%.
F. Observe on the Dev. Kit LCD that the displayed fan speed is 2350 rpm.

7 Experimentation: Hall effect sensor PWM response

A. Connect a 12 V supply to JP2.
B. Connect Fan 1A to JP4.
C. Connect a signal generator to JP1 pins 1 and 19 (GND) and 17 (+3V3).
D. Configure the signal generator to output a square wave of frequency 25 kHz, fixed duty cycle.
E. Observe at TP2 the resultant waveform.

8 Further verification: PWM DAC operation

A. Connect a 12 V supply to JP2.
B. Connect a 3.3 V supply to JP1 pins 1 and 19 (GND), 20 (+3V3).
C. Connect a signal generator to JP1 pin 3 (TIMER0_CC1) and ground.
D. Configure the signal generator to output a square wave of frequency 100 kHz.
E. Vary the duty cycle across an appropriately wide range of values.

F. Observe that the resultant voltage at JP3 for each duty cycle value is or is nearly equal to the supply voltage multiplied by the duty cycle.

9 Further verification: Modulated supply tachometer output

A. Connect a 12 V supply to JP2.
B. Connect a 3.3 V supply to JP1 pins 1 and 19 (GND), 20 (+3V3).
C. Connect a signal generator to JP1 pin 17 (TIMER1_CC1) and ground.
D. Connect Fan 1A to JP4.
E. Configure the signal generator to produce a square wave of frequency 25 kHz and with fixed duty cycle.
F. Observe at JP1 pin 4 (PCNT2_S0) that the voltage does not exceed the voltage of the 3.3 V supply.
G. Observe at JP1 pin 4 (PCNT2_S0) that the waveform is equivalent to the waveform which can be observed at TP2.

10 Further verification: fan speed control

A. Modify the third mode in the program specification (see section 14.4.3) to include at least four duty cycle variants.
B. Connect a 12 V supply to JP2.
C. Connect a 3.3 V supply to JP1 pins 1 and 19 (GND), 17 and 20 (+3V3).
D. Connect Fan 1A to JP4.
E. Set the Dev. Kit power source select switch to "DBG."
F. Connect through USB 1 the Dev. Kit to a computer.
G. Configure the Dev. Kit to operate in the third mode (see step A).
H. Connect Dev. Kit expansion header pin 13 (PB12 / LETIM0_OUT1) to JP1 pin 11 (+3V3 / TIMER1_CC2).
I. Cycle through duty cycle variants, performing steps J and K each time.
J. Observe at PB12 that the duty cycle of the waveform is as expected.
K. Observe at TP2 that the frequency of pulses is as expected.

F6.4 Action items: notes

Action items 1 and 2 verify the most basic functionality of the prototype—that a fan can be energised and can start correctly when connected to the prototype.

Action items 3, 4, and 5 verify that the microcontroller operates as intended in the various specified control modes without requiring the development kit be connected to the proof-of-concept prototype circuit.

Action item 6 verifies that the microcontroller correctly and accurately measures fan speed. The fan tachometer produces two pulses per rotation, and so the frequency of approximately 78.33 Hz (~4700 rpm) should be read as 2350 rpm.

Action item 7 determines the effect of a PWM-modulated supply voltage on the output of the Hall effect sensor contained within the fan. A core driver in the inclusion of the PWM DAC was that the effect of PWM on the output of the Hall effect sensor was unknown and so could not be relied upon. The results of this experiment could significantly impact the decisions made in the producing of any final design.

Action item 8 verifies that the PWM DAC operates correctly—that is, that the voltage across its output is correct. Action item 2 verified only that the PWM DAC could energise a fan, and so could be completed irrespective of whether the operation of the PWM DAC was correct (provided that the output voltage was at least the starting voltage of the fan).

Action item 9 verifies correct tachometer output, after its level shifting—that is, that the output signal is firstly transmitted through the shifter, is secondly of the appropriate level for provision to the development kit, and is thirdly free of noise which could be misinterpreted as a tachometer pulse.

Action item 10 verifies that the fan responds as expected to the PWM output the microcontroller produces.

F7 Test Results

F7.1 Basic verification: bare fan connection

12th January 2018

The proof-of-concept prototype was connected to a 12 V supply and a 3 V (rather than 3.3 V) supply, and Fan 1A was connected at JP4 (the bare fan connection), in the manner specified in action item 1.

Additional equipment used included a Fluke 115 digital multimeter, a Tektronix TBS 1042 oscilloscope, an Aim-TTi TG330 function generator, two Philip Harris 0–25 V college power units, and all the necessary cabling and probes. This equipment is shown in a photograph dated the 12th of January in Appendix F4.

The supplies were energised, but the fan was not observed to start as predicted in action item 1D. The supplies were de-energised and, in order to troubleshoot the issue, Fan 1A was disconnected from the prototype. Fan 1A was connected directly to the 12 V supply and was observed to start and run, and so it could be ruled out that Fan 1A was at fault.

It was suspected that MOSFET M3 was non-functional. In order to confirm this, continuity tests were performed. The first continuity test, between the fan connector 12 V pin and TP7, was positive—this was the expected result, as the 12 V line is not routed through the MOSFET. The second continuity test, between the fan connector ground pin and TP3 with 3 V supplied to the gate of the MOSFET, was negative—there was no continuity between the fan ground and the prototype ground. The MOSFET switches the ground connection, and so continuity between fan and prototype ground was expected when the MOSFET was in conduction.

This was taken as further evidence indicating a non-functional MOSFET, as the FDMS7682 is rated for a typical gate–source threshold $V_{GS(th)}$ of 1.9 V, with a rated maximum of 3 V. In order to confirm this, the MOSFET gate resistor R14 was probed during energisation to determine whether any current was flowing to the gate of the MOSFET. If current flow is present, the MOSFET could be taken as functional. This current flow would result in a voltage drop across R14, which could be measured during energisation as the MOSFET gate charged. With the probes in place, and with a reduced voltage of 2 V supplied to the MOSFET gate, Fan 1A was observed to start and continue to rotate on energisation. Further, when the circuit was energised without the probes in place, Fan 1A was observed to start and continue to rotate once the probes were placed across R14.

It was not determined whether the fan would continue to rotate once the probes were removed (having been put in place to start the fan). Initially, this was taken to be the case, as no discernible change in fan speed was observed immediately after the removal of the probes. However, when the MOSFET gate was de-energised, the fan continued to rotate for more than 30 seconds before coming to a halt. It may have been the case that this effect occurred after the removal of the probes, but too short a time elapsed between probe removal and the de-energising of the MOSFET gate to be able to confirm this. The MOSFET was not energised again after the two aforementioned tests which probed R14, as it was feared that a risk of irreversible damage to the MOSFET was present.

The cause of the effect where the fan continued to rotate for a considerable length of time after the apparent de-energising of the MOSFET gate is not known. It was considered that it is possible for the PWM DAC inductor to feed back into the circuit via its flyback diode—which would be the only complete conduction path for the inductor when no fan is connected—but, even if this were the case, the connection between the 12 V rail and ground (via a fan) is controlled by the MOSFET, and so no complete circuit could be formed without conduction through the MOSFET. However, if the MOSFET were subjected to conditions which would cause it to conduct without a gate current, it is expected that the MOSFET would be destroyed.

If the MOSFET were destroyed and failed closed, there would exist a continual connection to ground, and so the fan would be expected to start irrespective of a gate voltage being applied. This was not the case. If the MOSFET failed open, it would result in a permanent disconnection from ground (barring any extremely high voltage), and so the fan would be expected to never start. Again, this was not the case. The fan would start when R14 was probed, and so the MOSFET must be capable of some form of controlled operation.

One potential explanation is that the voltage at the MOSFET drain—possibly as a result of backfeeding from the PWM DAC inductor—was sufficient to cause the body diode the MOSFET incorporates to begin conducting. If this were the case, a complete circuit to ground could be formed without driving the gate of the MOSFET. The simplest method of determining whether this is the case would likely be to measure the voltage at the drain throughout the operation of the MOSFET, with particular attention paid when the MOSFET begins to conduct (i.e. while and after probing R14) and when the gate signal is removed.

Additional measuring points include TP4 (PWM DAC output), TP7 (additional power supply screw terminal 12 V), and the solder joints between resistors R10 and R11 (3.83:1 potential divider on the bare fan connection 12 V line). Each of these points enables measurement of the 12 V line on the prototype.

Before performing this test, it is necessary to determine why the MOSFET began to conduct when R14 was probed and whether doing so is likely to damage the MOSFET or other circuit components. As a possible start, the resistance of R14 could be measured to determine if R14 is faulty. If R14 is shown to be faulty, the connection of the multimeter across it may have provided a high-impedance but still sufficient path for current to flow to the MOSFET gate, which would not need relatively large currents to switch (as might be required for the operation of a BJT device). Further research is therefore required.

18th January 2018

A Maplin UZ82D simple analogue multimeter was temporarily acquired in order to perform basic confirmatory tests before the 19th of January, when further and more advanced testing could be performed in an electrical workshop.

A number of resistance measurements were taken to verify expected continuity and expected discontinuity. The results of these measurements are given in the following table.

Probe 1	Probe 2	Resistance
M3 (Source)	TP3	0 R
	TP7	40 K
M3 (Drain)	JP4 (GND)	0 R
	JP4 (12 V)	∞
M3 (Gate)	TP3	∞
TP7	JP4 (12 V)	0 R

For clarity, in this table JP4 refers to the bare fan connection with that reference designator, and M3 refers to the MOSFET with that reference designator (which is connected at its drain to the GND pin on JP4 and at its source to circuit ground). JP1 is the 20-pin expansion header with that reference designator.

TP3 is the screw terminal (JP2) ground test point, TP4 is the PWM DAC 12 V test point, and TP7 the screw terminal 12 V test point. No other test points were used.

In all cases, these results verify what was anticipated. Although the measurement of 40 kiloohm may at first appear incorrect, it must be remembered that there is a potential divider permanently connected between 12 V and ground (formed by the resistors R10 and R11). Although the aggregate resistance of that potential divider is nominally 48.3 kiloohm, the distance on the multimeter scale between a reading of 40 and a reading of 50 kiloohms is on the order of millimetres.

Two further measurements between the source of M3 and the gate resistor R14 were taken to confirm gate signal continuity: between the source and the land pattern pad which should be directly connected (by a trace) to the source, and between the source and the land pattern pad connected to the source via the gate resistor R14. In the first test, the expected resistance of 0 ohms was observed. In the second test, an infinite resistance was observed. To further confirm, a reading of the resistance between those two pads was taken, and observed as infinite. To confirm correct technique, the resistance of R15 (the ostensibly identical resistor on the gate of MOSFET M4) was taken and observed to be approximately 500 ohms—a reasonable deviation from the expected 470 ohms, given that the meter in use is analogue.

This preliminarily confirms the suspicion that resistor R14 is non-functional and the theory that the multimeter provided a sufficient path to the gate to enable the switching on of the MOSFET. For full confirmation, these tests will be repeated on the 19th with a more accurate digital multimeter.

If R14 is fully confirmed to be non-functional, it may be necessary to attempt its replacement. However, whether an equivalently-rated and sized resistor is available on short notice is not known.

The resistances of R7 (PWM DAC driver MOSFET gate resistor), R8 (PWM DAC fan PWM control signal MOSFET), and R15 (bare fan connection fan PWM control signal MOSFET) were observed as approximately 500 ohms, and so the testing of these portions of the circuit is likely to be possible.

19th January 2018

The measurements taken on the 18th of January were repeated, using a more accurate digital multimeter instead of an analogue meter. The results are shown in the below table.

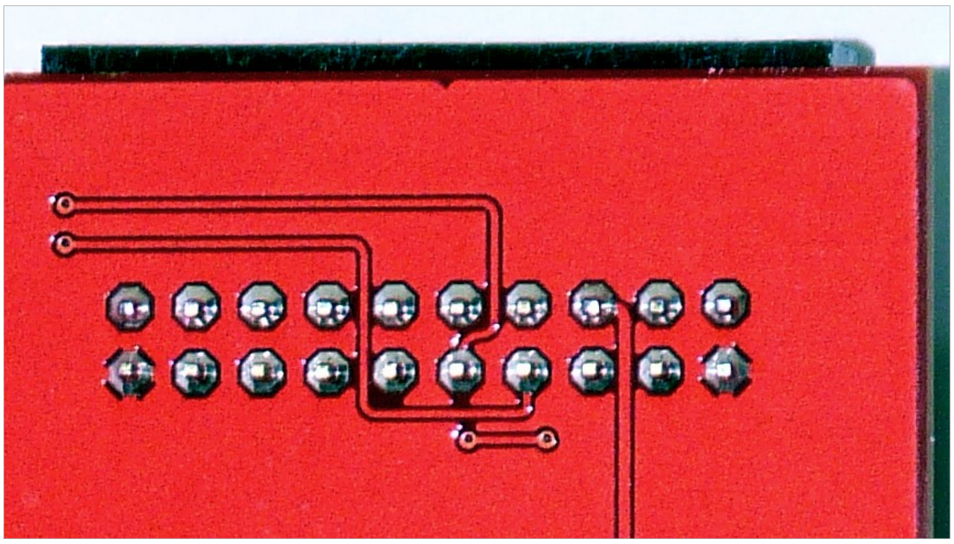
Probe 1	**Probe 2**	**Resistance**
M3 (Source)	TP3	0.1 R
	TP7	36.4 K
M3 (Drain)	JP4 (GND)	0.1 R
	JP4 (12 V)	23 M
M3 (Gate)	TP3	∞
TP7	JP4 (12 V)	0.2 R

A resistance of 0.1 ohms was reported when the probes were shorted together, and so values of 0.1 or 0.2 ohms are likely to be lower than reported.

Further measurements between the gate of M3 and the top pad of R14, and between the gate of M3 and the bottom pad of R14, were taken. As anticipated, for the first of those measurements a resistance of 0.1 ohms was reported. When the second measurement was taken, however, the reading was inconsistent—the multimeter alternated between an infinite resistance and around 472 ohms (the expected resistance). This would indicate, contrary to what measurements taken on the 18th appeared to reveal, that the resistor was functioning.

As the gate resistor was then most likely functional, the inconsistent reading was taken to indicate a poor connection between the resistor and its pads (e.g. as a result of poor solder joint formation). Although visual inspection did not reveal any obvious defects, the use of significant pressure when probing R14 did give a consistent reading. This was interpreted as the added pressure causing contact to be made where, in the absence of pressure, it would not otherwise be made. As final confirmation, considering that poor solder joint formation is unlikely given the assembly was performed by a professional service and likely involved the use of a reflow oven, a resistance measurement between the gate and a via after R14 was performed, and gave a consistent reading of 472 ohms. This consistent value indicated that a poor solder joint was unlikely—if the joint was poor and causing inconsistent measurement, a measurement would be inconsistent irrespective of whether it was performed at the resistor or at a connected location. Although the precise cause of the inconsistency in measurement is not known, it may be the case that a non-conducting layer of solder flux residue prevented measurement, and any added pressure scraped away part of that layer.

In performing this final confirmation, because it was suspected that solder joints may be the issue, a number of other components were probed. In particular, the ground on pin 1 of the 20-pin expansion header was probed with the resistance between it and TP3 measured. This resistance—which should have been around zero ohms—was measured as being infinite, apparently indicating an issue. In the course of taking this measurement, however, one of the probes was accidentally moved into contact with pin 2 instead of pin 1, where—with the other probe still in contact with TP3—a resistance of less than an ohm was measured. Clearly, this indicated a problem—on the schematic in Appendix F2, pin 2 on the expansion header is floating, unconnected to any portion of the circuit. A visual inspection of the expansion header solder joints was performed to determine whether any immediately obvious defect existed. Below is a photograph of those joints.



It can be seen that, while no obvious solder joint defects are present, there is an irregularity—while figure 9.1 in the development kit user's manual shows that the pins at either end on the bottom row are ground connections, it can clearly be seen that the pins at either end on the top row are connected to the ground plane. For the avoidance of doubt, "top" and "bottom rows" here mean the rows on the expansion header face, and not the board connections. The position of the board connections is reversed relative to the expansion header face—the bottom row of connections in this photograph corresponds to the top row of connections on the face of the expansion header.

For the further avoidance of doubt, this means that the bottom row is connected to where the top row should be, and the top row to where the bottom row should be. It is not clear how this occurred—the most likely explanation is that diagrams illustrating connection were misinterpreted. For example, figure 9.2 in the user's manual, if it were taken as a view facing towards the expansion header, could be interpreted as indicating that the ground pins were on the top row and not the bottom. Further, the numbering included with the expansion header footprint on the circuit board layout (see Appendix F3) could reasonably be interpreted as the pin numbers of the top or the bottom row depending on whether the numbers were taken to be the numbers of the pins visible from a top-down view, or the numbers of the pins closest to the surface of the circuit board. In any case, the issue is a design fault and should have been noticed during checks.

There are few potential fixes for this issue. Perhaps the simplest method, the expansion header could be de-soldered from the top of the prototype, and then re-soldered onto the bottom side. There is some risk of damage in soldering, but this is likely to be minimal.

Alternatively, it might be possible to use a solderless breadboard intermediary in a manner comparable to a telephone switchboard. This is perfectly workable on the proof-of-concept prototype side, as a 1 mm^2 solid conductor can easily be inserted into both the expansion header and a breadboard. However, on the side of the development kit, this fix is less workable. The development kit's expansion header is a set of pins, not receptacles, and so there cannot be attached any cable without means of securing that cable in place. The development kit does include so-called "breakout pads" (plated holes without components mounted in them), a number of pins exposed by the expansion header are only available on surface pads (rather than plated holes) located on the reverse of the development kit. It may be possible to reroute certain functions to the additional connections made available through the pads (pins PA12–14, PB9–10, PC7, PD8, PD13–14, PE0–3, and PF8–9), but use of these breakout pads would require either that cables be directly soldered to these pads, or that suitable receptacles be sourced and also soldered to the pads.

While direct soldering of cables to the pads would be the fastest method, risk of damage is increased relative to the use of a standard plastic receptacle. However, it may not be possible to source a suitable receptacle in sufficient time. It may also be the case that the required microcontroller functions (in particular, those exposed on pins PC3–6, which are present on the reverse of the kit) cannot be routed to the receptacles. If this is the case, full testing of the prototype may then require the use of a single set of functions, reconnected as appropriate, to control all aspects of the prototype. This would be relatively cumbersome, but would be a preferable alternative to a complete inability to test the prototype.

Returning to the carrying out of action item 1, once the source of the issues was known, the simple relocation of conductors to the suitable locations enabled the bare fan connection to be tested.

Using the same equipment as on the 12th of January, a 12 V supply was connected with a 3 V gate signal applied to pin 18 (instead of the pin 17 specified in the action item). The measurements specified in action item 1E to 1G were not taken.

26th January 2018

The measurements specified in action item 1E to 1G were taken using the same equipment as on the 12th of January. The supply voltages were set around 12 V and 3.1 V. The results were as follows.

At TP6, a voltage of 120 mV was observed. When the supply voltage was 8 V and not 12 V, this voltage decreased to 80 mV. These measurements are equivalent to fan currents of 0.25 A and 0.16 A, the former being the expected value and the rated current of the fan. While no predictions were made about the expected fan current at lower voltages, this reading does give some insight—if the decrease in fan current were linear, it would be expected that the ratio of voltages and the ratio of currents would be equal. Instead, they are marginally different—the ratio of 120 mV to 80 mV is 1.5, while 0.25 A to 0.16 A is 0.15625. This would indicate that, as expected, a fan controlled by modulation of the supply is likely to require some form of control loop rather than a simple "set and forget" system where the voltage can be set and the speed can be assumed correct.

At TP2, the voltage was observed to be around 7.11 V. This is below the expected maximum of 12.6 V, and around the fan rated starting voltage of 7 V. This is also consistent with the value observed at TP1 during completion of action item 2.

At JP1 pin 10 (the output of the potential divider at the bare fan connection), the voltage was observed to be 2.51–2.52 V. This is below the maximum of 2.6 V.

This action item is completed.

F7.2 Basic verification: PWM DAC fan connection

19th January 2018

The proof-of-concept prototype was connected to a 12 V supply and a 3 V (rather than a 3.3 V) supply, and Fan 1A was connected to the PWM DAC fan connection, designated JP3, in the manner specified in action item 2.

The equipment in use was the same as that used on the 12th of January in carrying out action item 1. Adjustments were made as appropriate to account for the issue of the connections to the expansion header being incorrect. Specifically, ground connections were instead made at pins 2 and 20, and the gate signal was applied on pin 4 (rather than pin 3).

The fan was observed to start and continue to rotate, as expected.

The voltage at TP5 more than 1 second after starting was observed as 1.4 V, not the 0.1 V that was expected. TP5 is the output from the current transducer. As the op-amp was not powered during this test, it is unlikely that this output can be assigned any meaning. It is possible that the op-amp could have been powered through its inputs, but it is unlikely that the voltage through these inputs would be sufficient to ensure reliable operation. The action item has been updated to include a step for connecting a 3.3 V supply.

If, after the op-amp is connected to the supply, a similar reading is produced, it will be necessary to undertake further investigation. Texas Instruments IN180A2 devices have a fixed gain of 50, and so—at the 0.25 A stated on the face of the fan—the voltage dropped across the 10 milliohm current-sense resistor should be around (0.25 amp) × (10 milliohm) = 2.5 millivolt which, multiplied by the gain, would result in an op-amp output of 125 millivolt. If the op-amp output remained at 1.4 volts, this would indicate that around 28 millivolts was dropped across the current-sense resistor, equivalent to a fan current of 2.8 A (well above the 1.5 A standard maximum for a fan). In this scenario, it would be necessary to confirm the true current drawn by the fan by measurement. However, other than at the supply screw terminal, the connection of an ammeter may be problematic. The ammeter, required in series with the fan, would need to be attached to the header by clips, which may be too large to attach to one pin of the fan header without coming into contact with another.

Otherwise, the voltages observed at TP1, TP4, and TP7 were within expected ranges. The voltage at TP1 was observed at approximately 7 volts (the starting voltage for the fan), and the voltages at TP4 and TP7 were observed as equal to the supply voltage (which would not normally exceed 12.6 volts).

The measurement specified in action item 2G was not taken.

26th January 2018

The measurement specified in action item 2G was taken, and the measurement in action item 2E was repeated with the op-amps powered.

The supplies were configured for approximately 12 V and 3 V. A voltage of 64 mV was observed at TP6, equivalent to a current of 128 mA. This differs from the fan rated current, and the current measured in carrying out action item 1E, of 0.25 A. The reasons for this are not clear, and no decrease in fan speed was visible during operation. More precise measurement of fan speed could be made, if desired.

At JP1 pin 6 (the output of the potential divider at the PWM DAC fan connection), the voltage was observed to be 2.502 V. This is below the maximum of 2.6 V.

This action item is completed.

F7.3 Basic verification: MCU PWM DAC control mode

19th January 2018

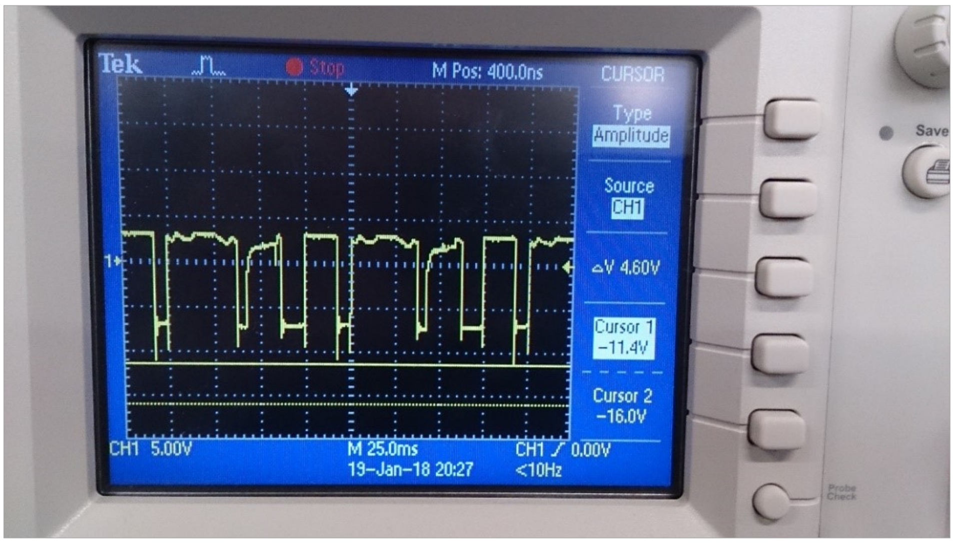
A preliminary test relating to the function of the PWM DAC was carried out. The microcontroller was not used as the basic verification in action items 1 and 2 had not yet been completed, and so it could not be guaranteed that the connection of the microcontroller was safe (primarily in terms of risk of damage to property).

The equipment in use was the same as that used on the 12th of January in carrying out action item 1. Adjustments were made, as appropriate, to account for the issue of the incorrect connections to the expansion header.

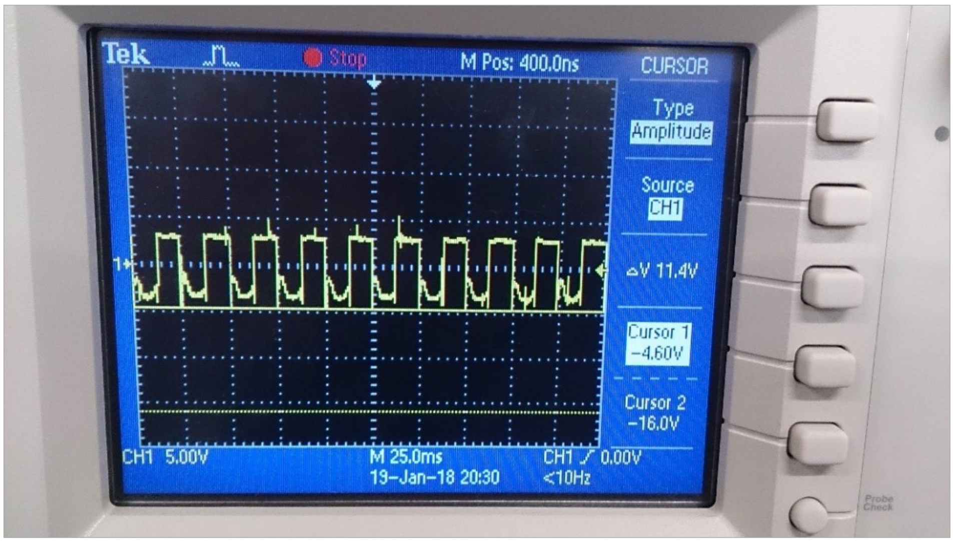
The proof-of-concept prototype was connected to a 12 V supply, and Fan 1A was connected to the PWM DAC fan connection.

For a first measurement of the signal from the tachometer, the prototype was also connected to a constant 3 V (instead of 3.3 V) supply. Measuring the output of the fan tachometer with an oscilloscope, there was observed to be a period of around 12 milliseconds between the positive edges of each pulse, giving a frequency of approximately 83⅓ Hz, equivalent to 2500 rpm—reasonably close to the rated fan speed of 2350 rpm.

Subsequent measurements of the tachometer signal were made with a function generator connected to the gate of the PWM DAC driver MOSFET M1. To drive M1, the function generator was configured to produce a square wave with a peak-to-peak amplitude of 6 volts, and a zero-volt D.C. offset. Photographs of the fan's tachometer output, as captured by an oscilloscope, are shown for each of the measurements made with M1 driven by the function generator.



Operation at 30 Hz

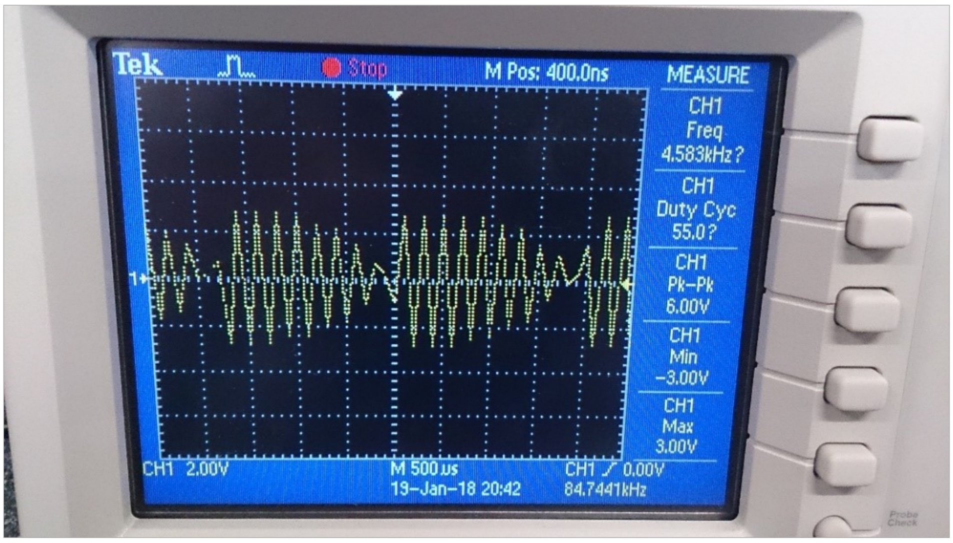


Operation at 25 kHz

As was expected at especially low frequency, the impact on the tachometer signal is significant. While a 30 Hz driver frequency was never a possibility, it was believed that the gathering of data at multiple, highly varying frequencies would allow for better analysis of any waveforms.

At 25 kHz, a recognisable square wave is present on the output. Although noise is present, especially at the very end of each negative edge, it may be the case that this noise is introduced by the probe (especially given its consistency). No careful selection of probes was made, and so the probes in use may be somewhat unsuited for measuring these signals.

A further test was attempted at 100 kHz—the planned operating frequency of the PWM DAC driver—but, at this frequency, the connected fan failed to start. At slightly lower frequencies, the fan would start again. Moving gradually from the starting frequency of 100 kHz to 0 Hz, the fan was visually observed to change speed. The probable cause of this behaviour was revealed when the output of the function generator was probed.



Signal generator output at 84.74 kHz.

At an output frequency of 84.74 kHz (a frequency below 100 kHz around which the fan began to rotate again), the supposed square wave produced by the signal generator was extremely distorted. As can clearly be seen, the waveform does not even vaguely resemble a square wave.

The issue of the connected fan not starting at 100.5 kHz likely stems from the effect such a distorted signal would have on the PWM DAC. The PWM DAC, if it functions correctly, should produce an output voltage roughly equivalent to the supply voltage multiplied by the duty cycle of the driver. That is, for a 10 V supply and an 80% duty cycle, an output of around 8 V would be expected. The effective duty cycle of this distorted signal, if it were sufficiently low and if the PWM DAC were to function even somewhat correctly, could result in a voltage below the fan starting voltage being produced. In particular, if the 55% duty cycle estimated by the oscilloscope were correct, the PWM DAC output could be near 6.6 volts.

If the distortion in the signal generator output increased with frequency, the added distortion at 100.5 kHz may have resulted in a lower still duty cycle and, as a result, a lower still output voltage. At 84.74 kHz, the approximately 6.6 volts may have been just close enough to the starting voltage for the fan to start. Indeed, performance at 84.74 kHz was severely reduced—the period between fan tachometer pulses was around 43 ms, equivalent to 23¼ Hz or nearly 700 rpm. This speed is only marginally different from the upper bound of the standard minimum fan speed for a 4-pin PWM-controlled fan, which is specified as 30% or less.

For reference, 30% of the 2350 rpm maximum fan speed is 705 rpm. If this is not a coincidence, it would indicate that the fan was operating at or near its minimum fan speed, and so that the fan was supplied at or just above its starting voltage, as would be expected when the PWM DAC is driven with a duty cycle of 55%.

Although no firm conclusions can be drawn from this experiment, the results are useful and can aid in the carrying out of future experiments. Firstly, the results appear to show that the PWM DAC is somewhat functional—the speed of the fan was around what would be expected at the starting voltage of the fan, and the duty cycle reported by the oscilloscope was around that which would be expected to produce a voltage near the fan's starting voltage.

Secondly, the results revealed that the function generator that was in use may not be reliable. While it is possible that the generator in use was faulty, if others of the same type are unable to produce well-formed square waves they are unlikely to be useful. The generator, an Aim-TTi TG330, does include symmetry control, which is equivalent to duty cycle control. That symmetry and duty cycle control were equivalent was not known during the experiment, and so no test of this functionality could be made. However, the function generator manual specifies that ratios from 10:1 to 1:10 can be set, and so it is possible that performance could improve if symmetry were enabled. If performance cannot be improved, the only practical solution (other than the use of another function generator) is to use the PWM generators from the microcontroller as drivers.

26th January 2018

As no computer was available, a deviation from action item 3 was made. The development kit was powered by a CR2032 button cell, and the development kit power source select switch set to "BAT." The multimeter and oscilloscope in use were those used on the 12th of January in carrying out action item 1.

The multimeter was connected across the positive terminal of the cell and a ground exposed as one of the development kit's breakout pads, and a voltage of the expected magnitude, around 3 volts, was observed. The microcontroller was also observed to start, with the text "IDLE" becoming visible on the LCD on the development kit.

Using the menu presented through the development kit LCD, the microcontroller was configured for the first mode of operation in the program specification (that mode being one where the PWM DAC is driven). The oscilloscope probe ground clip was attached to the shield of the USB micro-AB port on the development kit, and the probe manually held against pin 3 (PC0) of the expansion header (ensuring that no contact with other pins was made).

No output was observed on the expansion header pin. The breakout pad labelled PC0 was probed to confirm that this observation was not the result of operator error, and no output was observed. To determine whether the wrong pin had been probed, each of the pins on the expansion header was probed, and again no output was observed. The breakout pads were not probed in this manner, as the header pins and the pads largely expose the same connections (with the breakout pads exposing a small number of additional connections).

A fault in the probe was ruled out by two factors—first, that the oscilloscope was registering 50 Hz noise produced by mains wiring; and second, that the probe passed a probe check performed by the oscilloscope. The same probe had also been used earlier in carrying out action items 1, 2, and 7. One possibility is that the USB port was not grounded, although section 6.8 (p. 114) of the USB 2.0 specification states that "the shield must be terminated to the connector plug," and that "the shield and chassis are bonded together." It would then be reasonable to expect that a connection to circuit ground is present. This could not be verified as a continuity test would risk damage to the microcontroller.

The simplest possibility is that the firmware was incorrect—that, while firmware routines were correct, it did not configure the microcontroller output and so the output of the peripherals was never transmitted to the microcontroller pins. If this is the case, a simple addition to the firmware is likely to be all that is required to fix the issue.

28th January 2018

In order to determine whether the firmware was at fault in producing the result observed on the 26th of January, the example code included with Silicon Labs' application note AN0014 (2017) was reviewed. This application note, covering the timers included with EFM32 microcontrollers, includes an example detailing how to produce PWM output with the timers.

The example code, in the file `main_timer_pwm.c`, carried out the same actions as the prototype firmware, with one exception. In the `main` function, the example configures a microcontroller pin (PD1) for push–pull mode.[5] The example code also enables timer 0 capture/compare channel 0 output at location 3.[6]

From the EFM32WG990 datasheet (2014, p. 69), it can be seen that location 3 for timer 0 CC channel 0 is pin PD1. This indicates, as confirmed by the EFM32WG reference manual (2014, p. 760), that the peripheral must be configured to output on a specified pin, and that pin must also be configured for output. This latter step is not done by the proof-of-concept prototype firmware, and so this is the most likely explanation for no output being observed.

1st February 2018

The development kit was connected to a laptop and configured to operate in the first mode in the program specification so that the action item might be carried out. The equipment in use was the same as that used on the 1st of February in carrying out action item 8.

The breakout pad PC0 was probed, and no voltage was observed. In probing all breakout pads, a high voltage was observed on PB9 and PB10 (the pins connected to the development kit pushbuttons). To confirm that the code for setting the pin output level was functioning correctly, a test program which only set PC0 high was written, tested, and observed to work correctly.

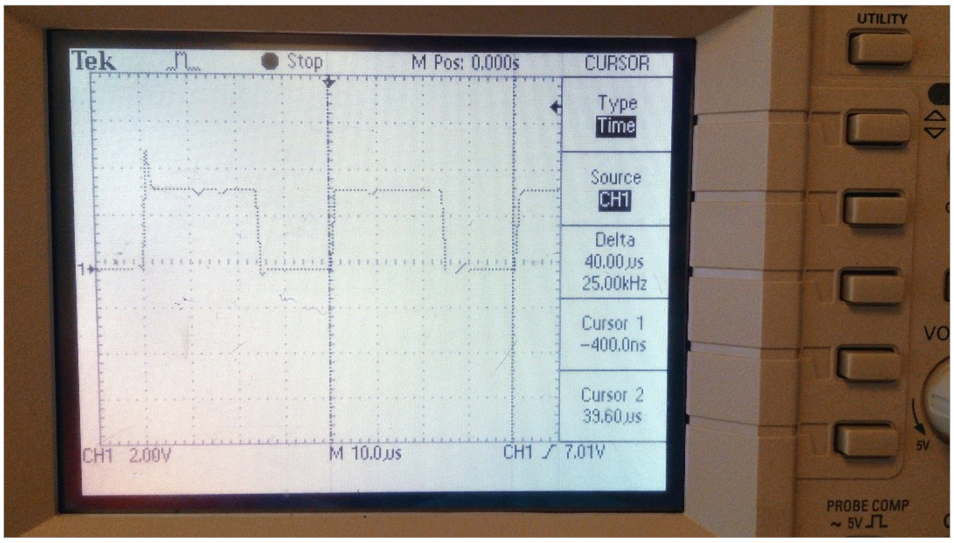
On rereview of the code for the firmware, it was discovered that the code which configured the microcontroller PWM generators had not been enabled. The code was enabled and the test reperformed.

A constant high voltage was observed at PC0. As the control modes start in the 100% duty cycle mode, this is the expected output. On actuating pushbutton 1 to switch to the next duty cycle variant, it was observed that the indicator message on the development kit LCD did not change until the pushbutton was actuated a second time. Subsequently, on probing PC0, a constant high voltage was present

[5] A mode where the output pin is connected to both logic-high and logic-low voltage rails by oppositely-doped transistors driven by a single base or gate signal. Thus, when the gate signal is present, on transistor is switched on and the other switched off, allowing control of whether the output is at logic high or low.

[6] To clarify, EFM32 microcontrollers include reroutable IO, and so a signal from a single peripheral can be routed to up to 7 output pins, numbered 0–6 and called "locations."

when the 60% duty cycle mode was indicated, and a PWM waveform when the 100% duty cycle mode was indicated. This desynchronisation between indication and output is likely related to the issue where a second actuation was required for a change in indicator to occur. The PWM waveform is shown below.



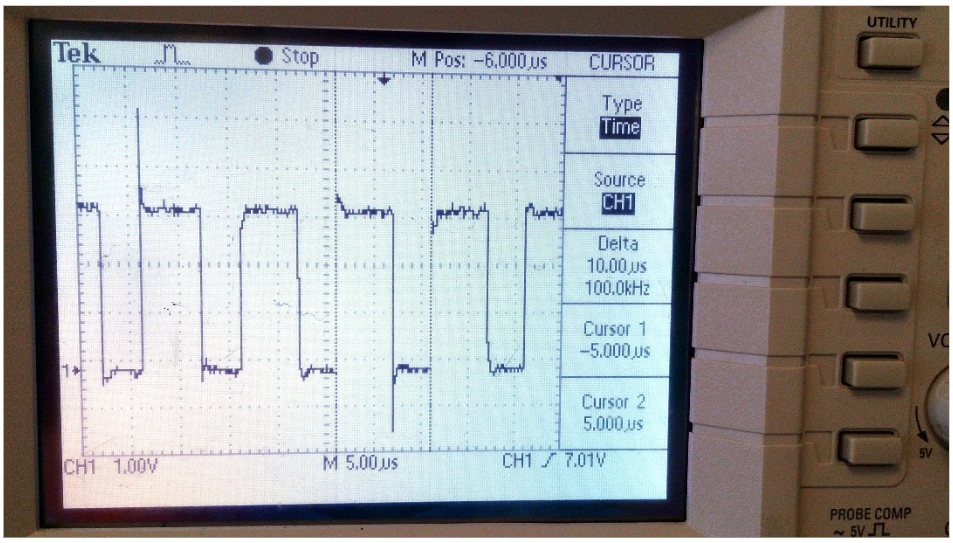
Microcontroller PWM output, first program mode.

As can be seen, the waveform produced has a 40 µs period and is held high for around 25.2 µs, giving a duty cycle of approximately 63%. While this duty cycle is largely correct—the output being expected to have a 60% duty—the frequency is not. Rather than the 100 kHz specified in the action item, the observed period gives the waveform a frequency of 25 kHz. While these may appear to be vastly different frequencies, 25 kHz is one of the two frequencies which are to be output by the microcontroller in the control modes, and so this points to an error in the configuration of the timers.

The first step taken in troubleshooting this was to adjust the divisor provided to a call to the `CMU_ClockDivSet` function in the setup portion of the state-handling code for the control mode. This divisor was adjusted from 1 to 4 and the output from the microcontroller probed, with no change observed. This indicates that, despite the documentation stating it will “set clock divisor/prescaler” (2017), the function silently fails if instructed to set the TIMER1 prescaler. It may behave in this way because the timer prescaler configuration is contained in a timer register and not a clock management unit register, but no statement to this effect could be found in documentation, and no timer library function (other than the `TIMER_Init` function) enables the setting of the timer prescaler. To confirm this, the prescaler setting in the `TIMER0_CTRL` register was manually adjusted, and the test reperformed after manually adjusting the prescaler to 1.

Before reperforming the test, as further verification, the configuration data for the timer was checked against equation 20.4 in the microcontroller reference manual (2014, p. 533). It was confirmed that the configuration data would produce the anticipated output frequencies.

The output observed at PC0 on reperforming the test is shown.



Microcontroller PWM output (corrected), first program mode.

It can now be seen that the output has a 10 µs period with 6 µs spent positive, and so both the duty cycle (of 60%) and the frequency (of 100 kHz) are correct.

PB11 was probed and observed to be at 0 volts, as expected.

This action item is completed.

The issue discussed above where two actuations of the pushbutton were required for the indicator to change, and where the wrong duty cycle was indicated, was, on review of the relevant code, determined to be the result of the variable in the code which kept track of state being assigned an incorrect value. In the code dealing with transitions between duty cycle variants, the state is switched on. For transition into the 60% duty variant, the current state must be the 100% duty variant. It is thought that, absentmindedly, the semantics of this check were taken as reversed (i.e. transitioning to, rather than from, 100% duty), and so the state variable was assigned the wrong value. This was corrected and the code verified to operate as intended. This issue did not affect the other control modes.

F7.4 Basic verification: MCU modulated supply control mode

26th January 2018

See the discussion for the 26th of January for action item 3. Equivalent actions were taken, with equivalent results.

28th January 2018

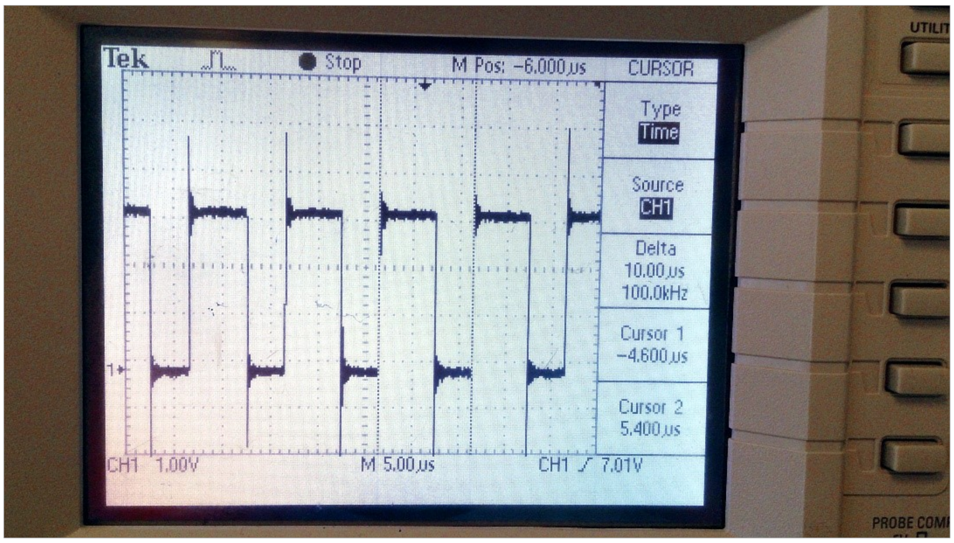
See action item 3's discussion for the 28th of January.

1st February 2018

See action item 3's discussion for the 1st of February. The issues and fixes set out for that action item also apply to this action item. This section discusses only the testing carried out after the fixes were made.

The development kit was connected to a laptop and configured to operate in the second mode in the program specification. The equipment in use was the same as that used on the 1st of February in carrying out action item 8.

The breakout pad PD7 was probed, and a constant high voltage was observed in the first duty cycle variant. Pushbutton 1 was actuated to switch the firmware into the second variant, and the indicator on the development kit LCD was observed to change accordingly. The following waveform was observed on probing PD7.



Microcontroller PWM output, second program mode.

As expected, a square wave of period 10 µs where the signal is positive for 6 µs, giving a frequency of 100 kHz and a duty cycle of 60%. These are the values set out in the action item and in the code for the firmware, respectively.

The output waveform observed here appears significantly noisier at its positive and negative levels than the equivalent waveform observed in action item 3. The presence of minor differences is to be expected—a difference timer is used here, with different circuit paths and routing, and with different silicon. This is unlikely to present any issues, but (not lining up with expectation) is notable.

At the breakout pad for PB12, a voltage of 0 V was observed.

This action is completed.

F7.5 Basic verification: MCU PWM control signal control mode

26th January 2018

See the discussion for the 26th of January for action item 3. Equivalent actions were taken, with equivalent results.

28th January 2018

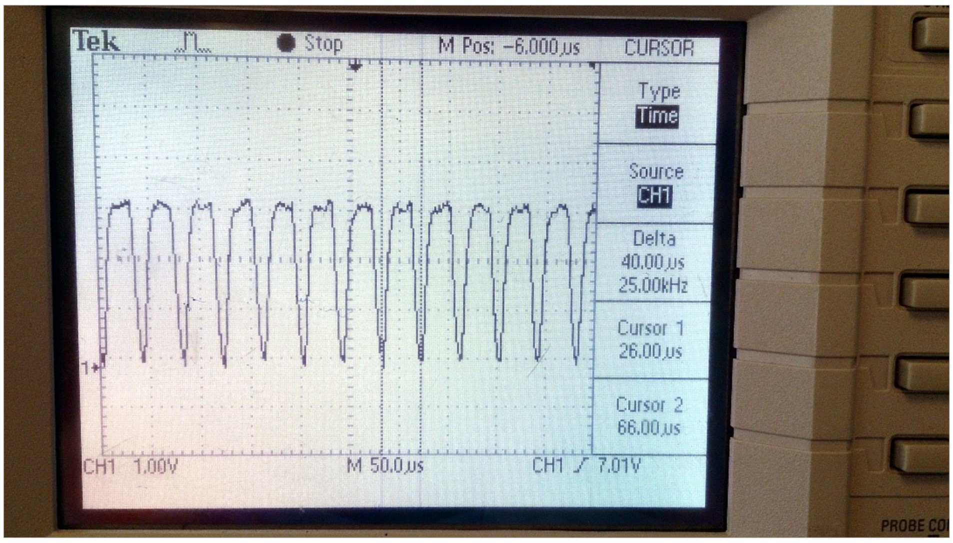
See action item 3's discussion for the 28th of January.

1st February 2018

See action item 3's discussion for the 1st of February. The issues and fixes set out for that action item also apply to this action item. This section discusses only the testing carried out after the fixes were made.

The development kit was connected to a laptop and configured to operate in the third mode in the program specification. The equipment in use was that used in carrying out action item 8.

A constant high voltage was observed at breakout pad PB12 on probing in the first duty cycle variant. On the actuation of pushbutton 1, the indicator displayed on the LCD on the development kit changed to indicate transition to the second duty cycle variant. On probing PB12 again, the following waveform was observed.



Microcontroller PWM output, third program mode.

It can be seen that the square wave shown above has a period of 40 µs. Owing to the shape of the wave, there was difficulty in determining the length of time the signal was high. At approximately 30 µs (or 75%), the output is likely at or near the desired 70% duty cycle.

The waveform is heavily distorted compared to those observed in carrying out action items 3 and 4, but this is likely attributable to the use of the low-energy timer—in working towards the aim of consuming less energy, it would not be unreasonable to anticipate some degradation in quality.

At PD7, a constant high voltage was observed, as expected.

This action item is completed.

F7.6 Basic verification: MCU pulse counting

1st February 2018

The development kit was connected to a laptop and configured to operate in one of the control modes set out in the program specification. As each control mode uses the same code for pulse counting, the specific mode was not recorded. The equipment was that used in carrying out action item 8 on the 1st of February.

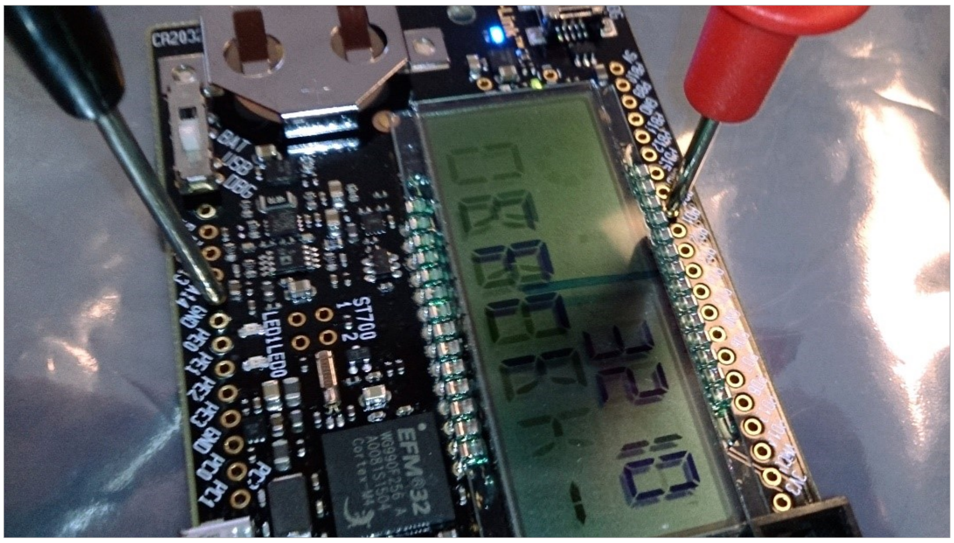
In programming the microcontroller on the development kit with the firmware, the code used for fan emulation was disabled. Accordingly, when placed in the control mode, the speed reported on the LCD on the development kit was 0 rpm.

A function generator was configured to output a square wave with a frequency around 106.7 Hz (equivalent to 3200 rpm). The wave was set to have a peak-to-peak amplitude of 2.2 volts, with a D.C. offset of 400 mV. This brought the peak voltage comfortably above the logic-high threshold of 2.1–2.3 V.[7] The function generator was connected to the development kit by contacting two probes with the broken-out pads for ground and a pulse counter input.

No change in the reported speed was observed on the development kit LCD.

Silicon Labs' application note AN0024 (2013) was reviewed, revealing that the inputs to the pulse counters must be configured as inputs through the registers for configuring GPIO, and that the microcontroller's internal alternative function routing also applies to inputs. Appropriate changes were made to the firmware, and the microcontroller was programmed with the updated firmware.

The test was reperformed, with the following indication observed.



Microcontroller reporting speed with 106.7 Hz input.

The output shown—a reported fan speed of 3210 rpm—is as close a result as is possible with the basic pulse-counting code in the firmware. The pulse counter is read 60 times a minute and, as two pulses represent a full rotation, the speed is the half the number of pulses multiplied by 60. Multiplication being commutative, this is equivalent to the number of pulses multiplied by 30, and so each pulse will represent an increment of 30 rpm. As 3200 rpm is not an integer multiple of 30, and as the frequency is 106.7 Hz (i.e. closer to 107 Hz than 106 Hz), the reading could be expected to vary between 3180 and 3210 rpm (the two nearest integer multiples of 30), with 3210 rpm appearing more often than 3180. This behaviour is what was observed.

While action item 6 called for use of a 78.33 Hz signal, that value was arbitrary. It was more convenient, given that the function generator had no readout and the oscilloscope readout precision was limited, to select 106.7 Hz. This choice also

[7] The microcontroller datasheet (Silicon Labs, 2014, p. 19) specifies a logic-high threshold voltage of 0.70×V_{DD}. V_{DD} is estimated as 3.0–3.3 V on the development kit, giving the values used above.

provided confirmation, discussed above, of the behaviour anticipated for when a non-integer tachometer frequency is present.

This action item is completed.

F7.7 Experimentation: Hall effect sensor PWM response

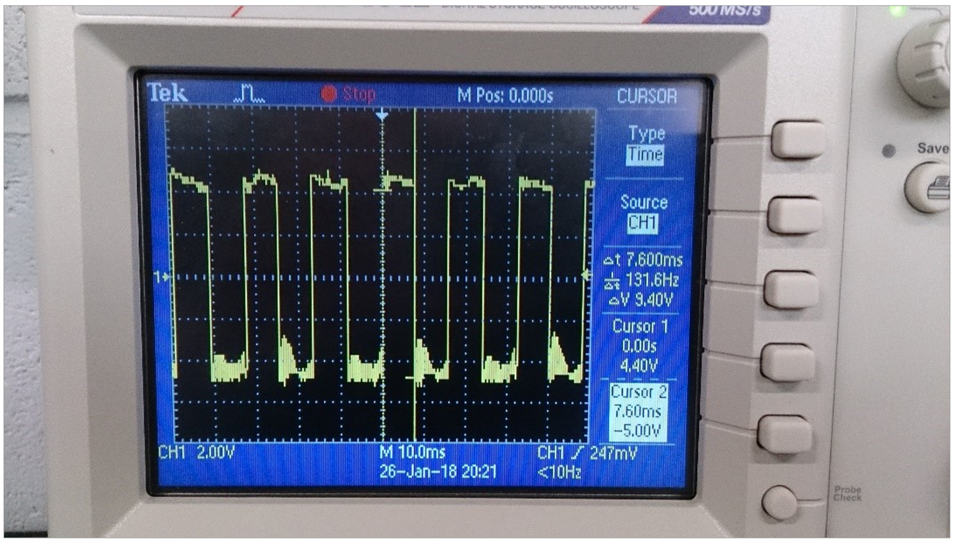
26th January 2018

As mentioned in the discussion for action item 3 for the 19th of January, it was observed that the waveform produced by the function generator was severely distorted under load. This appeared to result in a signal with relatively low duty cycle, and so it was necessary to determine whether the function generator could be used in testing the prototype. What was not known on the 19th of January was that the function generator included a “symmetry” function, described by the data sheet for the generator as enabling the varying of the mark–space ratio from 10:1 to 1:10, and what effect this would have on the apparently distorted waveform. The function generator was configured to produce a ~100 kHz square wave, with a peak-to-peak amplitude of 6 V, and was connected directly to an oscilloscope. The symmetry function was enabled, and the function generator response at three points over its range observed.

It was observed that the duty cycle of the signal changed as expected, but also that the D.C. offset of the signal changed with the changing duty cycle. At 50%, the wave was symmetrical around zero—that is, its maximum voltage was +3 V and its minimum voltage −3 V. At 10% duty cycle, this became +5 V and −0.8 V. When configured for 90% duty cycle, it became +0.8 V and −5 V. There is no risk to the MOSFETs (which are rated for ±20 V gate-to-source), but does present a practical problem in that, at higher duty cycles, the positive peak will not be great enough to switch the MOSFET on. There is no risk to the gate resistor—its rated resistance of 470 ohms means that, even at 6 volts positive, it would only dissipate around 77 mW, well within its 125 mW rating. It is expected that this could be corrected through application of the function generator’s variable D.C. offset function, albeit at an increased time cost.

In order to carry out this action item, the proof-of-concept prototype was connected to a 12 V supply, a 3 V supply, and a function generator as specified in action item 7. The equipment used was the same as that used on the 12th of January in carrying out action item 1.

The function generator was configured to produce a 25.12 kHz square wave, with a peak-to-peak amplitude of around 6 volts, and a duty cycle near 50%. TP2 was observed using an oscilloscope, and a waveform captured (shown below).

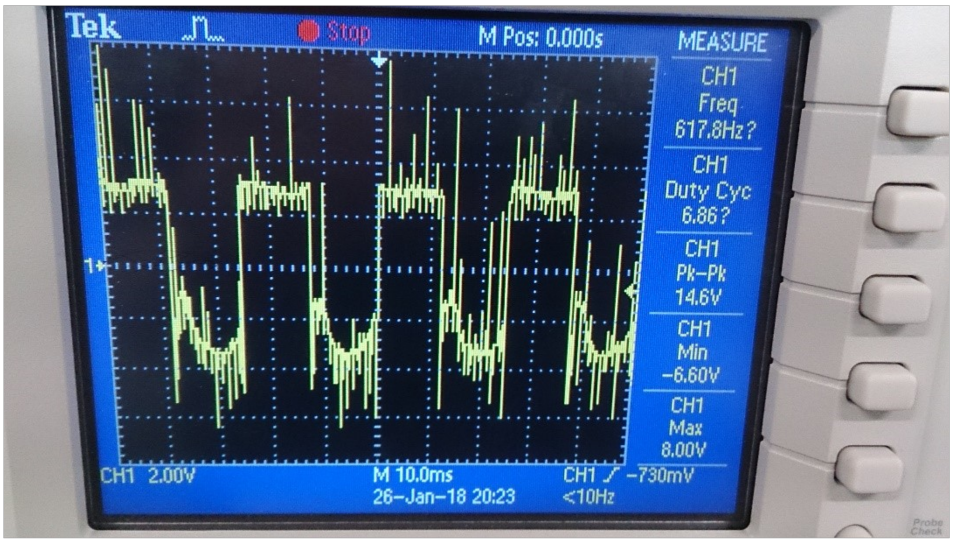


Tachometer output, supply modulated at 25.12 kHz.

As can be seen, some noise is present—in particular, at the positive and negative levels on the waveform. There can also be seen a repeating pattern of noise, with the same "shapes" repeated on the negative level every two pulses. This noise is most likely noise inherent to the fan—firstly, the negative-level noise repeats every two pulses, and a full rotation of the fan is represented by two pulses (and so the same pattern is unlikely to repeat unless it results from the fan itself); and secondly, noise which was largely the same was observed on the 19th of January in carrying out action item 3.

While it could be the case that the noise is caused by modulation of the supply and that the filter did not adequately "smooth" the supply voltage (which would cause the same noise to appear in that test), this does not mesh with the behaviour of the PWM DAC observed on the same day (where the fan ceased to run when the duty cycle of the signal driving the MOSFET was in the region of the starting voltage for the fan). At the same time, the fan connected to the circuit as it was configured for the carrying out of this action item did not cease to run until the duty cycle fell below around 42%. That is equivalent to an average voltage of 4.8 V, but it may be the case that the 12 V absolute voltage enabled rotation after the average fell below the rated starting voltage.

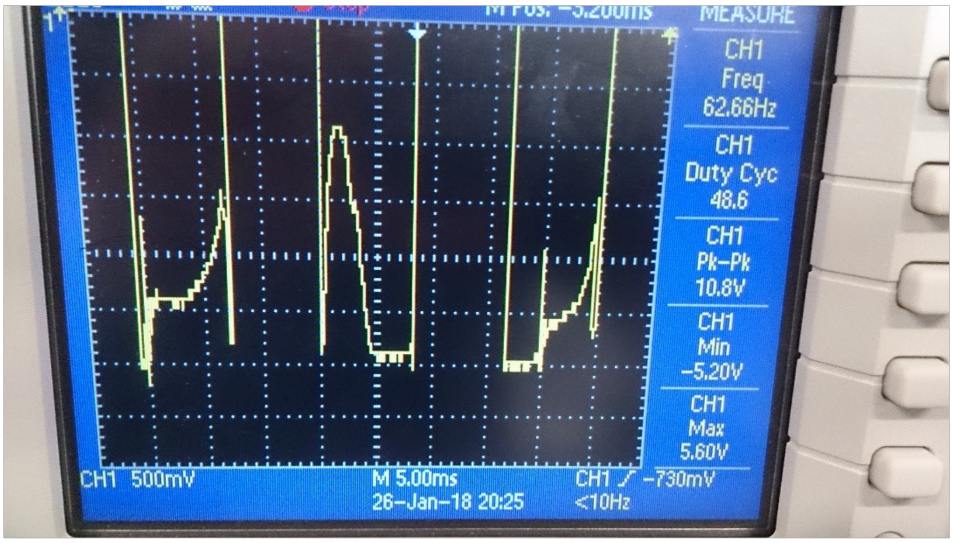
In part to determine the extent of the supply modulation on noise, and also in order to confirm whether the theory that a higher supply modulation frequency would reduce noise in the output (by reducing the length of each individual period where the fan's Hall effect sensor was deprived of current), a further test was performed at 99.33 kHz (instead of the 25.12 kHz specified in the action item). An image of the output in this configuration is given below.



Tachometer output, supply modulated at 99.33 kHz.

As can be seen, while the noise in the output has significantly increased, the basic noise present in previous waveforms remains present and largely unchanged in this waveform. This would appear to further confirm that the noise seen at the negative level, and the more minor noise at the positive level, is inherent to the fan, and so not an effect of the modulated supply.

This has promising implications for any final design—at a modulation frequency of 25 kHz, the output of the tachometer appears usable, and so it may not be necessary to include a PWM DAC in a final design. This in mind, the function generator was again configured for 25 kHz operation so that the spike in voltage at the negative level could be more closely examined. This spike is shown below.



Tachometer output (magnified), supply modulated at around 25 kHz.

The voltage spike shown in the middle of the above image was chosen from a reading for the reason that it has a particularly large magnitude, and so is likely a good indicator of the maximum spike which might occur (although there is likely to be some variation between different fans). If a voltage spike is small enough, it may be the case that no filtering is needed—the tachometer output is fed into a level shifter, and so the spike would need to be sufficient for the level shifter to operate for it to be transmitted to the microcontroller. The level shifter, a Texas Instruments CD74HC4050, includes in its D.C. electrical specifications a listing of the voltages considered high and low at 25 °C, reproduced below.

Parameter	Supply	Minimum	Typical	Maximum
High Level Input Voltage	2.00 V	1.50 V	—	—
	4.50 V	3.15V	—	—
	6.00 V	4.20 V	—	—
Low Level Input Voltage	2.00 V	—	—	0.50 V
	4.50 V	—	—	1.35 V
	6.00 V	—	—	1.80 V

Although no value is given for the supply voltage of around 3 volts used here, the values given for 2.00 V and 4.50 V can be considered. These values, considering that the spike in the image could be as great as around 2.50 V, place the spike either firmly in the high level region (with a 2.00 V supply) or in the unspecified region between 1.35 V and 3.15 V (with a 4.50 V supply). Even considering the smaller spikes in the image, the voltage could foreseeably reach 1.50 V, again enough to be placed in the high level or unspecified region for the level shifter.

For this purpose, the unspecified region may be treated as the high level region. As the output cannot be predicted, the worst-case scenario is that a high output is produced each time. In any final design, then, the effect of these spikes must be mitigated—the most correct solution likely being to use a transient voltage suppression diode, although a potential divider which reduces voltage by a fixed proportion could also be a practical solution if that proportion can be set such that the output remains in the high level region even under reduced supply voltage conditions.

This action item is completed.

1st February 2018

Additional minor work was done to determine whether the noise present on the readings presented previously in this section was inherent to the fan. Connecting the fan to a constant 12 V supply, the tachometer output was probed. Equipment used to do this was the same as was used on the 1st of February in carrying out action item 8.

The tachometer output observed did not include this noise, as can be seen in the image provided in the discussion for the 1st of February for action item 9.

F7.8 Further verification: PWM DAC operation

1st February 2018

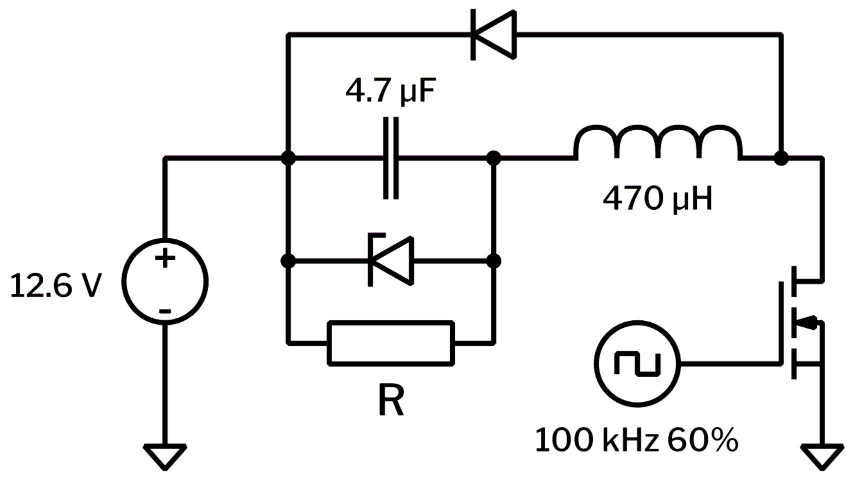
The proof-of-concept prototype was connected to a 12 V supply, a 3 V supply, and a function generator configured to output a square wave with a peak-to-peak amplitude of 2.5 V, where the negative level was approximately 400 mV. Precise voltage control was not possible due to the nature of the function generator.

The equipment is use was a Black Star 3225MP multimeter, a Tektronix TDS 220 oscilloscope, a Feedback FG601 function generator, and two Thurlby Thandar PL310 precision laboratory D.C. power supplies. This equipment is shown in the photograph dated the 1st of February in Appendix F4.

The function generator was first configured for approximately 100.2 kHz. As no control over duty cycle was supported, the waveform produced had a fixed duty cycle of 50% (as confirmed by oscilloscope observation). In this configuration, a voltage of around 11.8 V was observed at JP3—approximately equal to the supply voltage. It is possible that the supply voltage–observed voltage difference was the result of the method of measurement, as the multimeter had previously been observed to produce inconsistent (and potentially highly variable) readings when a firm connection was not made. In this case, measurement was taken by firmly contacting the multimeter probes with the appropriate pins on JP3 and, while the reported voltage was not varying, this factor should not be entirely discounted.

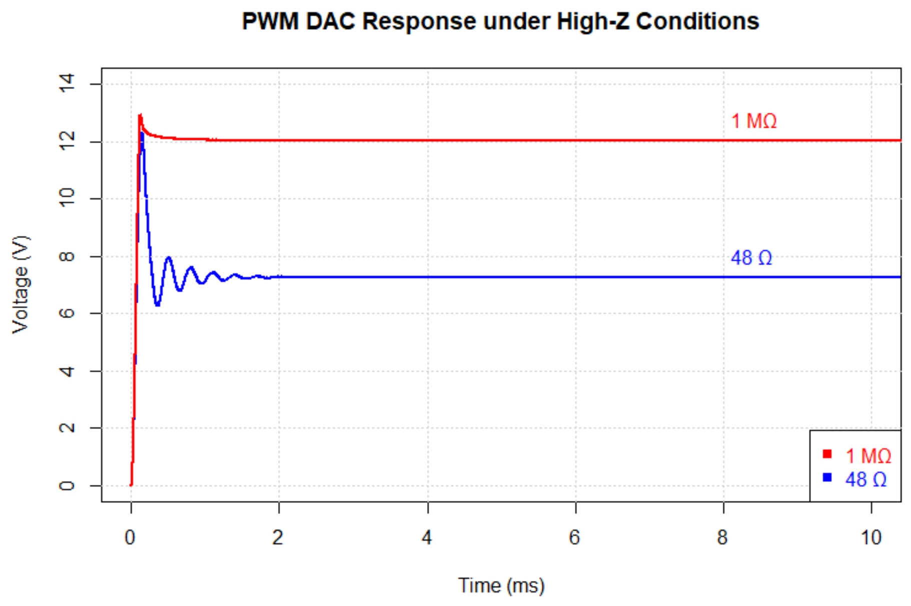
An equivalent measurement was performed with the function generator set to output a square wave of equivalent amplitude at frequency 24.8 kHz, with the same result—a voltage equal or near equal to the supply voltage was measured across JP3.

In each case, the expected voltage measured across the 12 V and ground pins of JP3 would be equal to the supply voltage multiplied by the duty cycle—with duty of 50% and a 12 V supply, this would be a voltage of approximately 6 V—not, as observed above, a voltage nearly equal to the supply. This would, at first, appear to indicate that the PWM DAC was non-functional, despite the previous apparent functioning observed on the 19th of January in carrying out action item 3. It is, however, worth considering that the large impedance presented across JP3 by the multimeter was likely not dissimilar to open-circuit conditions, under which correct operation is unlikely (by virtue of the filter comprising two parts linked by the load which are disconnected from each other when open-circuited). To confirm this, the PWM DAC circuit was simulated as below.



In the first simulation, resistor R was set to 48 ohms to produce the 250 mA which Fan 1A is rated to draw. In the second simulation, a resistance of 1 MOhm was used to represent the high-impedance input to a multimeter. No datasheet for the Black Star 3225MP could be found, and so this resistance is based on an example value given in an application note published by Fluke (2007). A plot showing the voltage across R in each simulation is shown.

The code used to draw the plot is given in Appendix C5.



As expected, the PWM DAC functions correctly for a resistance of 48 ohms—the voltage of 7.25 V is 60.4% of 12 V, corresponding to the 60% duty cycle (although a practical circuit is unlikely to be quite as closely corresponding).

Further, as predicted, the output voltage under high-impedance (and so effective open-circuit) conditions does not vary from the supply after stabilisation.

In order to complete this action item, then, the test must be carried out with the PWM DAC on-load. However, this action item cannot be completed until either a function generator capable of variable duty cycle control is sourced, or until it can be confirmed that the connection of the proof-of-concept prototype to the development kit is safe.

8th February 2018

The proof-of-concept prototype was connected to a 12 V and a 3 V supply, and Fan 1A was connected to JP4 to enable on-load testing. The equipment used was the same as that used on the 1st of February.

An oscilloscope was connected to both supplies. Fan 1A was de-energised and the waveforms displayed by the oscilloscope were observed. No obvious impact on either supply was observed. It was determined that it would be safe to connect the development kit to the proof-of-concept prototype to operate the MOSFETs.

In order to help determine whether the fan speed response behaviour observed on the 19th of January in carrying out action item 3 was the result of the quality of output produced by the function generator, the PWM DAC driver MOSFET was first operated using the output of a function generator. The generator in use, a Feedback FG601, was known to be capable only of producing an output wave with a duty cycle of 50%. At approximately 25 kHz, the fan operated at visibly reduced speed and a low whine could be heard coming from the fan. Probing the exposed metal on the reverse of the fan connector, 12 V was observed across the 12 V and ground connections of the fan, and the output of the tachometer was observed at 26.32 Hz (equivalent to around 790 rpm). Increasing the frequency towards 100 kHz, the fan slowed to a near stop at 66 kHz. A low whine persisted until the frequency was raised above 74 kHz. The fan did not begin to move again at 100 kHz.

The PWM DAC was then driven at 100 kHz from the development kit. At duties of 100% and 60%, the fan reported speeds of 2050 rpm and 1042 rpm—equal to around 50.8% of the former speed. This gives further credibility to the suspicion that the failure of the fan to operate at higher frequencies was related to the quality of function generator output.

In testing at 25 kHz, driven from the low-energy timer, the fan did not start in the first duty cycle variant (intended to be 100%). Reviewing the firmware, the timer was configured to operate at 100% duty by setting its duty to one above the upper bound value. This is valid with normal timers, but (as noted in Appendix F5.2) the low-energy timers do not have special handling for this configuration. The fix was simple, and involved setting the duty to the upper bound value. This may not have been noticed in carrying out action item 5 as the constant 0 V and the constant 3.3 V output would both appear as a constant straight line on the output of an oscilloscope, with the only difference being their relative elevations (which could be only a number of millimetres at larger scales).

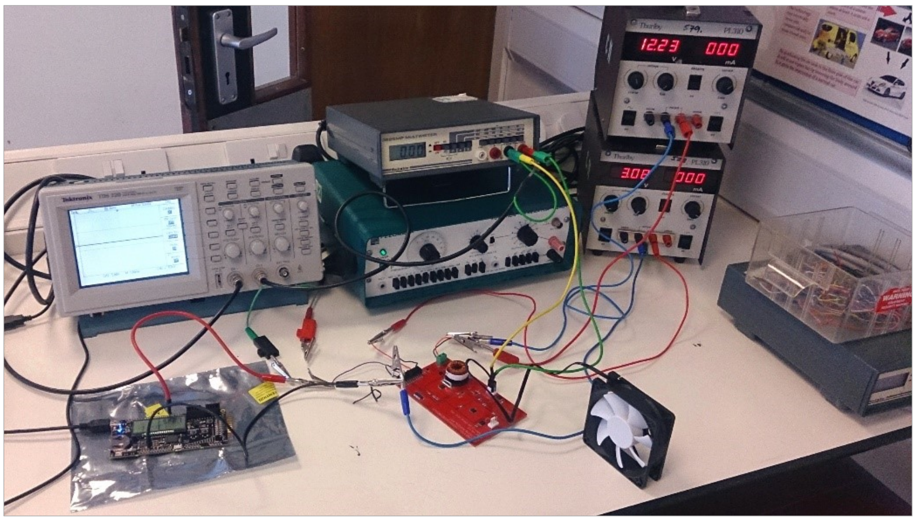
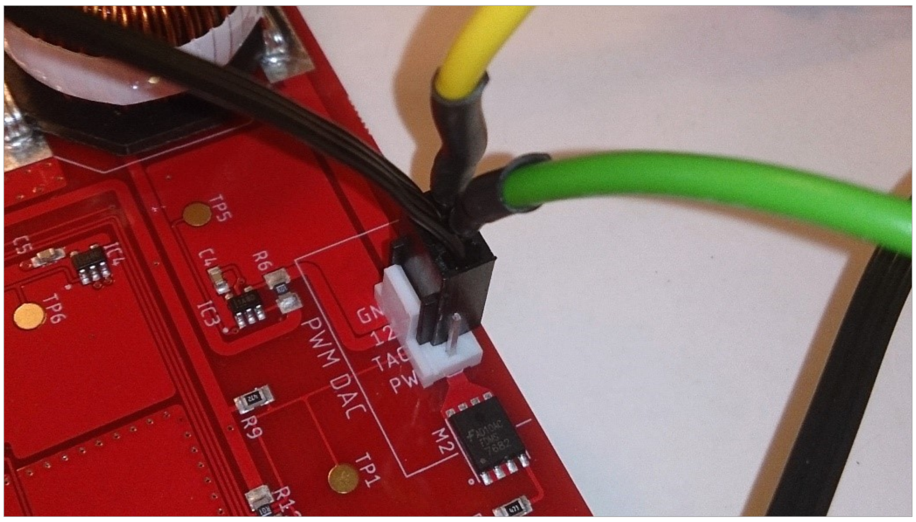
The basic operation at 25 kHz and 100 kHz confirmed, the development kit was cycled through a number of duty cycles at each frequency. The output from the kit and the output from the tachometer were recorded, and are given below.

Expected		Observed			
		25 kHz		100 kHz	
Duty	RPM	Tach.	RPM	Tach.	RPM
100%	2350 rpm			68.49 Hz	2055 rpm
80%	1880 rpm			51.28 Hz	1538 rpm
70%	1645 rpm	47.62 Hz	1429 rpm		
60%	1410 rpm			36.50 Hz	1095 rpm
40%	940 rpm			21.74 Hz	652 rpm
30%	705 rpm	25.00 Hz	750 rpm		
20%	470 rpm	11.52 Hz	346 rpm		
15%	353 rpm			0 Hz	0 rpm

While the speeds given in the table do not line up with the expected values, they do line up if adjustments are made using a speed at 100% duty of 2055 rpm. When operating at 80% duty, the 1538 rpm is around 75% of 2055 rpm, 1095 rpm 53%, and 652 rpm just under 32%. The PWM DAC output was not expected to align exactly with duty cycle, and had been observed in simulation to have a response which varied with load, and so these 5–8% differences are not cause for concern. It is also reasonable for the fan speed drop-off to accelerate the closer the output voltage becomes to the starting voltage.

Operating at 25 kHz, the speeds reported more closely aligned with the adjusted expected speeds: 1429 rpm is equivalent to 69.5%, 750 rpm 36.5%, and 346 rpm 16.8%. The precise reason for this is not known, but there may be a clue in the somewhat lower operating current observed in carrying out action item 2 on the 26th of January—the property of inductors to oppose changes in current may have greater effect at 100 kHz, where the faster switching speed could lead to reduced time for current to increase before the driver MOSFET switched off.

It being confirmed that the fan speed would change in response to the PWM DAC being driven with different duties, it was then necessary to investigate whether the 12 V observed across the fan at 790 rpm was accurate. Jumper wires were inserted into the fan connector and those wires connected to both a multimeter and oscilloscope, as shown below.

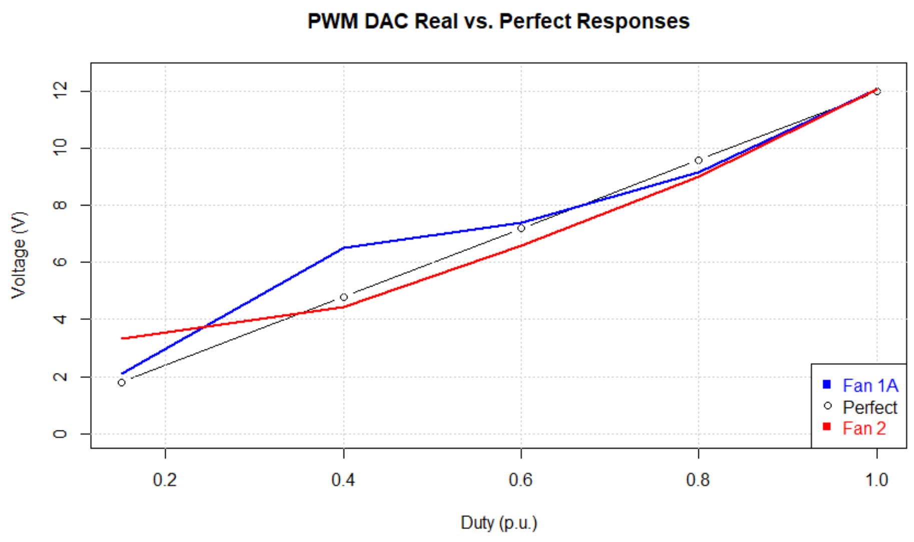
Jumper wires inserted into the fan connector.

In this configuration, both Fans 1A and 2 were operated at 100 kHz and cycled through a range of duties. At each duty, the voltage reported by the oscilloscope and the multimeter were recorded. The results are given in the below table.

	Fan 1A		Fan 2	
Duty	Oscilloscope	Multimeter	Oscilloscope	Multimeter
100%	12.10 V	12.10 V	12.00 V	12.10 V
80%	9.12 V	9.18 V	8.90 V	9.03 V
60%	7.40 V	7.40 V	6.60 V	6.58 V
40%	6.50 V	6.53 V	4.40 V	4.45 V
15%	2.06 V	2.09 V	3.28 V	3.31 V

Note that, where a varying waveform was observed, the mean voltage is given in the above table. The precision of the oscilloscope was limited by its display scale, and so where the last fractional digit of an oscilloscope measurement is zero, this has been added for formatting purposes.

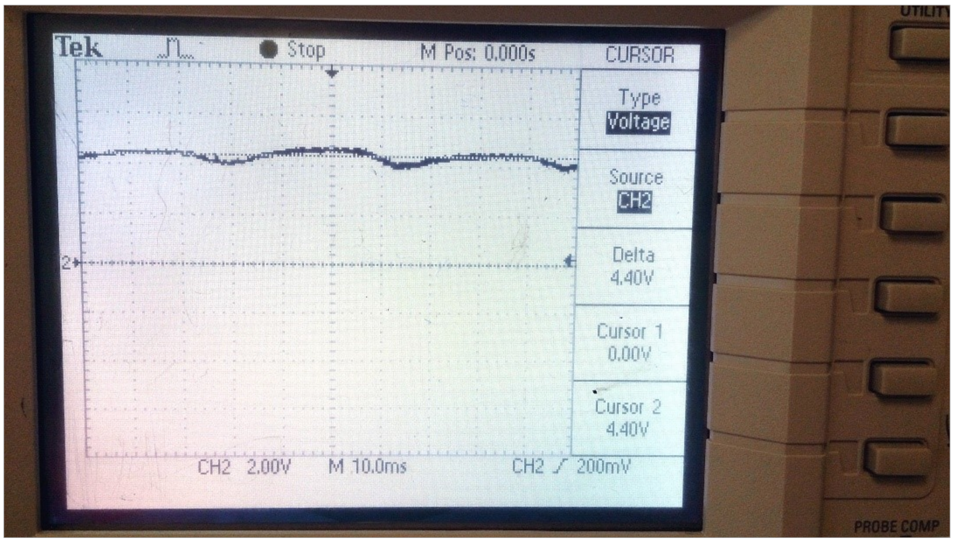
These values are plotted below, with the “perfect” line being the product of the supply voltage and the duty.



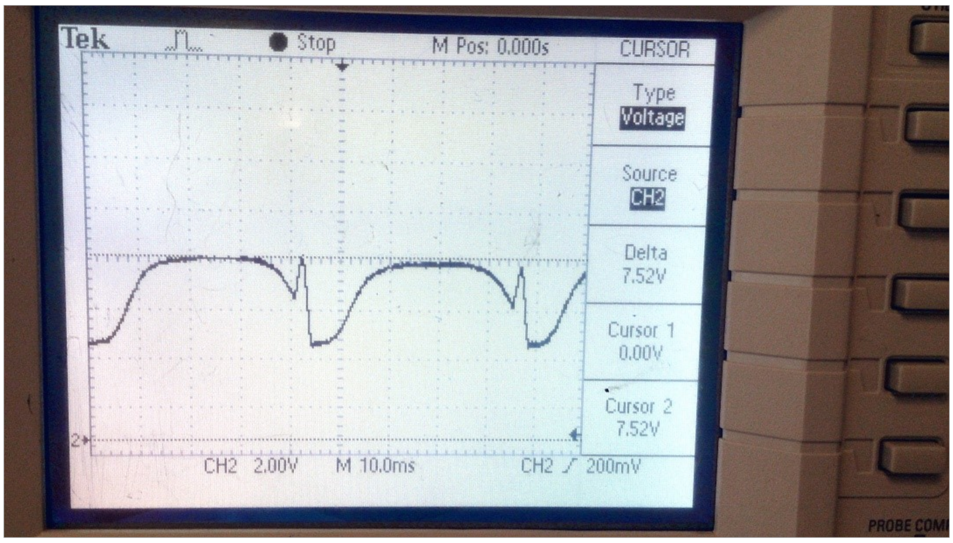
As can be seen, the responses generally approximate the perfect response. The response for Fan 2 falls slightly below the response for Fan 1A, a factor in which is likely the lower load presented Fan 2—a rated current of 60 mA instead of 250 mA—prompting an over-damped response, although the responses become more aligned at higher duties (potentially indicating that other factors are at play).

Both responses become less consistent, and diverge from the perfect response to a greater degree, at lower duties. Contrary to what could be expected, that the lower voltage (and hence lower effective load) would produce an overdamped response, both responses rise above the perfect response (where before, at higher duties, they had generally been in line with or below it). The response for Fan 2 in particular appears to plateau between 40% and 15% duty cycles, which could indicate that there exists a minimum voltage below which the PWM DAC cannot produce a response that corresponds to its duty cycle input. This behaviour is not observed with the response for Fan 1A, although—considering the variability that was previously observed with changing load—this may be related to the different currents drawn by the fans. The circuit was simulated (as on the 1st of February) to determine whether this behaviour could be reproduced, but the output in the simulation remained almost exactly linear with duty. It may be that the observed behaviour is dependent on the non-ideal characteristics of the components.

Weight is added to the suspicion that a factor in this behaviour is the load on the PWM DAC when the output waveforms are observed. The waveforms for each of the fans at 40% are shown below.



PWM DAC output with Fan 2 at 40% duty.



PWM DAC output with Fan 1A at 40% duty.

Neither of these waveforms is entirely steady, but it can be seen that the second waveform, for Fan 1A, is considerably less steady—varying between 3.84 and 7.52

volts, with a mean of 6.3 to 6.7 volts. The precise cause for this behaviour has not been identified, and the wave does not appear to represent the charge–discharge curve of an inductor. The peak-to-peak amplitude of the waveform of 3.76 volts is also far greater than the predicted (by simulation) voltage ripple of 13 mV. One possibility is that the change in inductance with frequency had a significant effect on the output, but, in this scenario, it would be expected that the damping of the response would change, rather than there being produced a waveform that does not decay and is a consistent shape, as is seen in the image.

This waveform could be caused by the interaction of the PWM DAC capacitor with the PWM DAC inductor, but only the rising edge of each pulse resembles a capacitor charge–discharge curve. The falling edge does not represent the curve for either a capacitor or inductor. This possibility is therefore less likely.

An interesting portion of the waveform is the final spike. It would make intuitive sense for this to represent the switching off of the MOSFET—causing the initial fall in voltage as the MOSFET falls out of conduction—and the operation of the inductor, resulting in the voltage rising before the flyback diode operates (which produces the fast falling edge). As can be seen in the image, this occurs over the period of around 12 milliseconds—8 milliseconds while the voltage is falling but before it spikes, 2 milliseconds where it is rising in the spike, and around 2 more milliseconds where it sharply falls to its lowest point before a new pulse starts.

The decrease in the voltage, if it were caused by the inductor discharging into the circuit in opposition to the supply, would be expected to occur far quicker—with an inductance of 470 μH and a circuit resistance of 48 ohms (the resistance which produces 250 mA at 12 V) plus 119 milliohms (the inductor inherent resistance), a time constant of just under 10 microseconds would be expected. Instead, the time for the decrease to occur is on the order of milliseconds. It is therefore unlikely that the drop and subsequent spike are caused by the operation of the MOSFET and the diode.

Ultimately, this great variability is not hugely important—it appears only to occur at relatively low duty cycles and so relatively low voltages, where the fan would not normally be operated. Using the perfect response, a duty cycle of 40% would produce a voltage of 4.8 volts, well below the advertised fan starting voltage. Even at the 6.5 volts seen in testing, the output remains below the starting voltage. If it were possible given the time constraints, a next step would be to investigate the response with varied capacitance and inductance in the circuit.

This action item is completed.

F7.9 Further verification: Modulated supply tachometer output

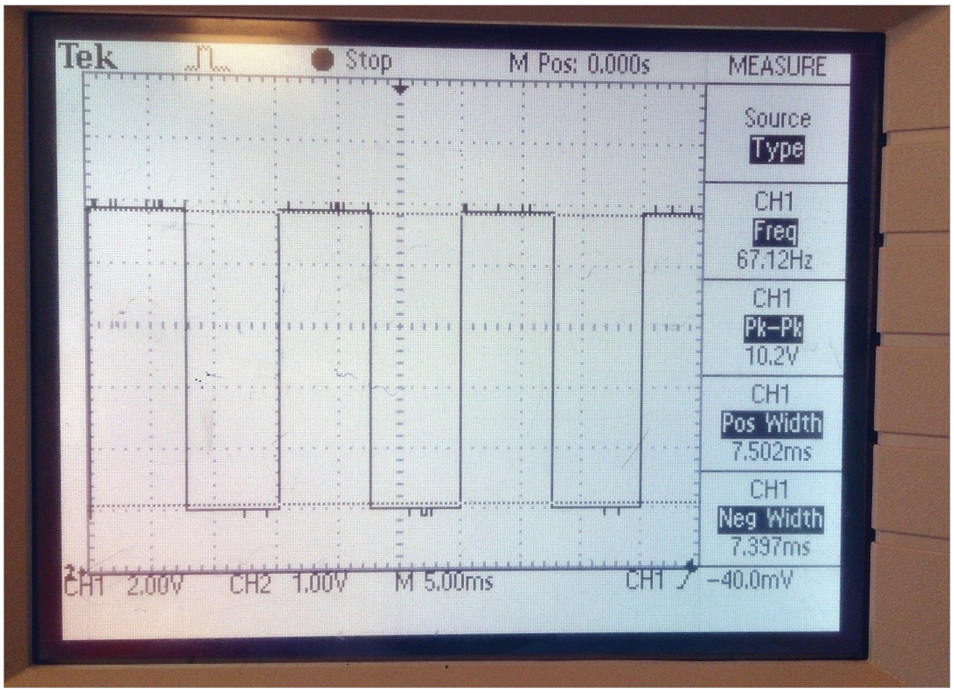
1st February 2018

The proof-of-concept prototype was connected to 12 V and 3 V supplies, and a function generator configured to output a square wave of frequency 24.9 kHz was connected to JP1 pin 17. Fan 1A was connected to JP4. The equipment used was the same as that used on the 1st of February in carrying out action item 8.

At TP2, the expected output from the fan tachometer was observed.

At JP1 pin 4, the voltage was observed to not exceed the supply voltage. However, a waveform equivalent to that observed at TP2 was not present. Instead, there was observed a constant high voltage. To confirm this reading, the schematic diagrams in Appendix F4 and the level shifter datasheet were reviewed, and the level shifter pins probed directly. This confirmed the readings.

In troubleshooting, the tachometer pulse frequency being the cause of this issue can be immediately eliminated. Each pulse produced by the tachometer lasts on the order of milliseconds, while the "high-speed CMOS logic" 744050 in use is specified as having rise and fall times not exceeding 1000 ns, propagation delay not exceeding 130 ns, and transition times not exceeding 110 ns (Texas Instruments, 2005, pp. 3-4). While no setup or hold times are given, even if the maximum switching frequency had a period double that of the sum of the listed times (that is, 2 × 2350 ns = 4.7 μs), that frequency would be 213 kHz—well above the hundreds of Hertz that might be output by a fan tachometer. The tachometer output recorded in the course of carrying out further work which related to action item 7 was examined for further information (shown below).



Tachometer output when supplied with a constant 12 V.

The review of the device datasheet in carrying out action item 7 provided a clue to the answer—in carrying out that action item, the minimum logic high voltages had been compared to the magnitude of a spike in noise to evaluate whether the spike might be registered as a tachometer pulse. These minimums depend on the supply voltage, and are specified at 1.5 V, 3.15 V, and 4.2 V for supply at 2 V, 4.5 V, and 6 V, respectively. Assuming the logic-high level changes linearly in the region of unspecified supply voltages, and using the specified values to produce a linear function $V_{HIGH} = 0.6735V_S + 0.1439$, logic-high at supply 3 V lies around 2.16 V.

The above image shows a peak-to-peak voltage of 10.2 V. Taken in isolation, this could be interpreted as the tachometer output not reaching fully supply voltage, but the oscilloscope was configured in D.C.-coupled mode, and so the offset of the negative level from the vertical position arrow (visible at the very bottom and right of the plot) reveals the true issue. As confirmed by a one-division separation between the vertical position arrow and the negative level, the tachometer output is alternating between approximately 12 V at its peak and 2 V at its trough.

This offset negative level, given that the logic-high voltage of the shifter is likely to be slightly lower than the specified voltage, may not be sufficiently low for the level shifter to register a logic low. If this is the case, and it appears likely, the behaviour of the level shifter would be that observed—a continual logic high, with no transition to logic low.

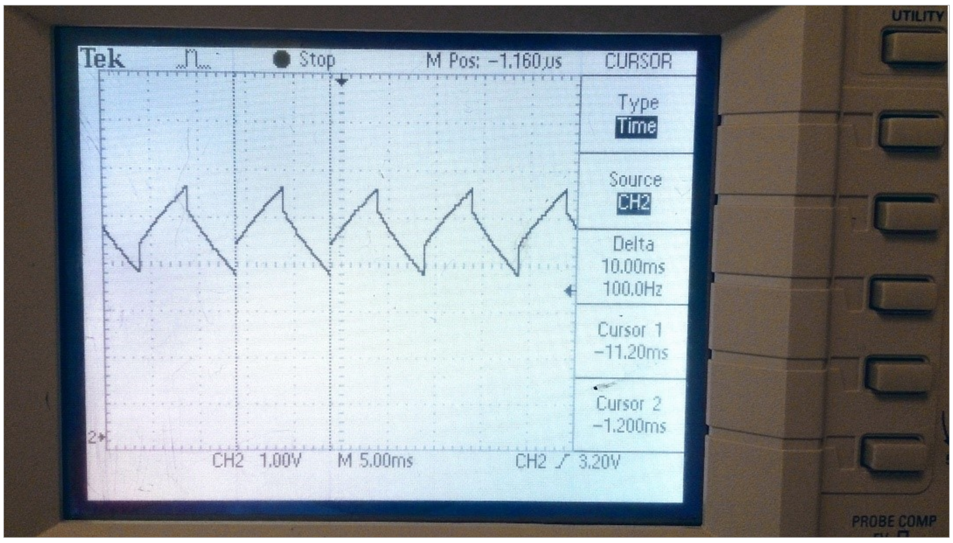
If this is the case, this issue could be fixed by any means which would sufficiently reduce the voltage to below the logic-high voltage, the simplest solution likely being the use of a potential divider. However, this has the potential to interfere with the pull-up to 12 V, and so any potential divider would need to be selected so as to present a higher resistance than the pull-up. This behaviour is also likely to vary from one fan to another, and so investigation with Fan 2 is desirable.

A suitable test verifying the operation of the level shifter would be to manually provide a signal, produced by a function generator and equivalent to what might be produced by a fan tachometer, and observe the output.

8th February 2018

The proof-of-concept prototype was powered with 12 V and 3 V supplies. The equipment used was the same as was used on the 1st of February in carrying out action item 8.

A function generator configured to produce a square wave of frequency 100 Hz, with peak-to-peak amplitude 11.6 volts and negative peak at 0 volts. An oscilloscope was connected to JP1 pin 4 to enable observation of the output of the level shifter. The function generator was connected directly to pin 5 ("2A") of the level shifter by contacting a probe to that pin. The output observed on oscilloscope is shown below.



Level-shifted 100 Hz, 0–11.6 V square wave.

The sawtooth-like waveform shown, while not the expected output, is no cause for concern on its own—a positive edge-triggered pulse counter should count a pulse of this shape no worse than it would a square pulse.

However, while the wave shape is not concerning, the amplitude is—the wave in the image has a peak-to-peak amplitude of 1.84 volts, offset from ground by 3.52 volts. Alternating between 3.52 and 5.36 volts, the signal would cause damage to the microcontroller if it were connected. In troubleshooting, no reason for this behaviour could be determined. The input wave had been confirmed to have the correct amplitude and D.C. offset, and probing pin 1 (“V_{CC}”) of the level shifter confirmed that approximately 3 volts was present. Further, while the datasheet only provides characteristics at 2, 4.5, and 6 volts, the device is specified for operation at 2–6 volts, and the first page of the datasheet mentions the device’s propagation delay at 5 volts. The specified output voltages, in each case listed, were below the supply voltage (as would be expected).

The failure of the level shifter is not critical—if the cause of the issue cannot be determined, another level shifter (or another method of level shifting) could be employed instead.

Fan 2 was then connected to JP4 to enable observation of its tachometer output for comparison with that observed for Fan 1A. The output of the tachometer at TP1 was observed to vary between 0 and 12 volts, rather than the 2 and 12 volts observed for Fan 1A. Any final design would therefore be unable to rely on any specific offset from ground, but would nonetheless be required to reduce signal levels to those usable by the level shifter.

This action item is completed.

F7.10 Further verification: fan speed control

8th February 2018

The firmware for the proof-of-concept prototype was modified such that the third mode in the program specification (see section 14.4.3) would cycle between duties of 100%, 70%, 30%, and 20%.

The proof-of-concept prototype was then connected to 12 V and 3 V supplies in the manner specified in action item 10. The development kit was connected to the prototype by jumper wires from its breakout pads, and Fan 1A to JP4. The equipment in use was the same as used for action item 8 on the 1st of February.

On configuring the development kit to operate at 100% duty, the fan was visually observed to be operating at a speed significantly slower than expected. The speed was observed to increase when the development kit was cycled to 70% (the next duty cycle), and observed to continue to increase through 30% and 20% duty. The response of the fan therefore appeared to be inversely related to duty. Reviewing the 4-wire PWM fan specification (Intel Corporation, 2005, p. 10), it is stated that the fan controller must “provide an open-drain [...] type output” (that is, an output which provides a controlled connection to ground). Any period where the gate for the PWM MOSFET is high, then, produces a low voltage on the fan side, and so it would make sense for the fan-side duty cycle to be the period where PWM MOSFET gate signal is low.

However, while this would make sense, whether this is the case is made less clear by the same page in the specification stating that the “signal is not inverted, 100%

PWM results in Max fan speed.” If the signal were not inverted, 100% duty would result in a constant connection to ground and so it would be expected that the fan would operate at minimum speed. At the same time, this statement could be interpreted as meaning the PWM “seen” by the fan—that is, the PWM produced by conduction of the MOSFET rather than present on the MOSFET gate. If this is the meaning, then 100% (fan-side) duty occurs at 0% MOSFET gate duty. Given the observed behaviour of the fan, it is likely that this was the intended meaning, and that (in implementing the firmware) this statement was misinterpreted.

This in mind, the test was reperformed and the tachometer output at each duty was recorded. Adjusting for the inverse operation of the fan, it remains possible to confirm correct operation. The results are given in the below table.

Duty	Tachometer	Inverted Duty	Equivalent RPM
20%	58.82 Hz	80%	1765 rpm
30%	50.00 Hz	70%	1500 rpm
70%	23.36 Hz	30%	701 rpm
100%	23.15 Hz	0%	695 rpm

Keeping in mind that the speed of a fan need only correspond to duty cycle ±10%, each of these results is within acceptable bounds (whether the maximum speed is taken as the rated 2350 rpm or the 2055 rpm observed whilst carrying out action item 8). As percentages of 2350 rpm, 1765 is 75.1%, 1500 is 63.8%, 701 is 29.8%, and 695 is 29.6%. If the maximum speed were taken as 2055 rpm, these percentages would be 85.8%, 73%, 34%, and 33.8%.

The result for 0% inverted duty would appear to indicate that 2350 rpm is the maximum speed, as each fan is required to have a minimum speed which is not greater than 30% of its maximum.

In terms of corrections, all that must be done is to “invert” the duty values in the code for the firmware. No further testing is considered necessary.

This action item is completed.

Appendix G
Ancillary prototypes

G1 USB prototype

The USB prototype has three core components—a hardware device supporting communication over USB, firmware for that device which implements the device class specification contained in Appendix D, and software which enables the host computer to interface with the device and enables a user to instruct the device.

This appendix covers the design of the prototype.

G1.1 Hardware: overview

The hardware device to be used is the development kit used in other portions of the project—a Silicon Labs EFM32WG-STK3800 development kit.

The microcontroller on the development kit includes a USB controller, and the kit itself includes a micro-USB port which can be used in the kit's connection to the host computer. Documentation for the USB controller is contained in the reference manual for the microcontroller, and in a number of application notes:

- AN0046 – USB Hardware Design Guide
- AN0052 – USB MSD[1] Host Bootloader
- AN0065 – EFM32 as USB Device
- AN0801 – EFM32 as USB Host
- AN0820 – EFM32 USB Smart Card Reader

While not all of the application notes are directly relevant, the information in the notes could be helpful in building an understanding of the controller.

Few other considerations can be made in relation to the hardware device. There are no changes which can be made to its design, and nothing that is required to be considered which would not fall under firmware or software.

G1.2 Firmware: overview

Microcontroller USB driver

The EFM32WG USB controller peripheral is relatively complex, with its control done through a set of 83 registers. Fortunately, Silicon Labs provides a library for basic control of the peripherals (called "emlib"),[2] a set of kit-specific drivers and a board support package, and "platform middleware" comparable to emlib but with APIs for more specific applications.

In particular, the USB platform middleware includes firmware stacks for use as a USB host and a USB device, as well as utilities common to both. This middleware can therefore be used in the USB prototype.

[1] Mass Storage Device (MSD), the name for the device class representing devices such as memory sticks, portable hard drives, and SD cards.

[2] "EM" being "Energy Micro," a company acquired by Silicon Labs which designed the EFM32 microcontrollers now sold by Silicon Labs.

USB identification

A USB device, as discussed in chapters 9.4 and 13.1 of the main body of the report, must identify itself to its USB host using a Vendor and Product ID. A Vendor ID can generally only be obtained by payment of a fee to USB-IF.

However, as also discussed, a number of USB licensees sublicense their ID, albeit on conditions (such as an upper limit on unit sales, or use of a specific product from the sub-licensor). One of these sub-licensors, John Otander (pid.codes), has allocated a VID–PID pair for test use.

That VID–PID pair (VID `0x1209`, PID `0x0001`) is available for private test use on the condition that it not be used on "any device that will be redistributed, sold, or manufactured." This pair can therefore be used in testing the USB prototype.

Integration with proof-of-concept firmware

Under the requirements specification set out in chapter 15.2 of the main body of the report, the USB prototype firmware component is to be created, to the extent practical, in such a manner as would enable integration of it with the firmware produced for the proof-of-concept prototype.

It is expected that the most significant design consideration resulting from this is the need to design the USB prototype firmware around the use of a periodic timer rather than interrupts. That is, the firmware would need to be designed so as to permit periodic checking of the USB controller peripheral, rather than the instant reaction provided by interrupts.

It is likely to be the case, for example, that too long of a waiting period will result in the operating system "timing out" the USB device and treating it if it had been disconnected. For example, the Microsoft Windows USB Core Team have stated at various points in a blog post (Microsoft Corporation, 2009) that a device will be disconnected if it "times out," albeit without any specific time period given. A degree of trial and error is anticipated here, but—given that a USB device will be polled at a maximum of 1000 Hz—a timer period of a handful of milliseconds is likely to be sufficient.

Operating system-specific descriptors

As the above-referenced blog post mentions, the Microsoft Windows operating system supports (and will request) a number of Windows-specific descriptors. In chapter 11 of the main body of the report (which dealt with research relating to the driver stack to be used), under "firmware considerations," it was noted that Windows supports the automatic loading of WinUSB on the connection of a USB device. This is achieved by returning a particular Windows-specific descriptor.

If WinUSB is to be used—and it must be, as the alternative is the development of a bespoke kernel mode driver, which would offer no advantage at significant cost to time available—it is necessary for the USB prototype to respond with one of these descriptors on interrogation.

These descriptors are set out in the Microsoft OS Descriptors Specification, of which two versions exist: version 1.0 (2007), specified for Windows XP, Server 2003, Vista, and Server 2008 onwards; and 2.0 (2017), specified for Windows 8.1 and 10 onwards. In order to support the automatic loading of WinUSB, the v2.0

specification must be used. Conformance with the v2.0 specification may require the firmware to report that it implements USB version 2.1—the Binary Device Object Store descriptor is specified as part of the USB 3.0 specification, which requires that v2.1 be reported for "Enhanced SuperSpeed operating in one of the USB 2.0 modes" (USB-IF, 2013, § 9.6.2). Confirmation of whether this is required can be obtained by testing.

G1.3 Firmware: discussion and remarks

Microcontroller USB driver issues

As discussed in Appendix G1.2, Silicon Labs provides a driver enabling use of the USB controller peripheral on its EFM32WG microcontrollers. Ostensibly, this is utilised in a codebase by the typical method—for SiLabs' Simplicity Studio IDE,[3] through an "import" function which includes a link to driver source files provided with the IDE. This link makes it so that the source files are logically present in a codebase despite actually being somewhere else in the filesystem.

This import initially appeared successful, but on compilation an error with the imported files was reported: `usbconfig.h`, a C header file the driver expects a user to provide to describe the particular USB application, could not be found. A simple error to fix, all that was required was that the relevant driver source files be copied to a location with a `usbconfig.h` file. This is the fix that was applied, and the error message did not appear on compilation.

On compiling after applying this fix, however, a large number of separate errors were reported in `em_usb.h`, the top-level header for the driver which exposes the definitions required to interact with the driver. In that header, errors stated that the types `uint8_t` and `uint16_t` could not be resolved. This is not an issue that would be expected, as these type names are standardised. Generally, this would indicate that the correct header file—`stdint.h`—had not been included, but including that header did not solve the issue. Examining the contents of that header, the C macros `__int8_t_defined` and `__int16_t_defined` (which are used to conditionally compile in code translating the compiler-defined types into the standard types)[4] were indicated as being undefined. Examining the contents of the `machine/_default_types.h` header, however, appeared to indicate that those symbols were defined, and so that the definitions should have been present and compiled in. The reason for their not being compiled in is unclear.

For clarity, the IDE indicates lines which are conditionally compiled out with a grey background, and code after `#ifdef __int8_t_defined` was greyed while code defining that symbol was not.

The root cause of this issue not appearing readily fixable, an additional header file masking `stdint.h` was added which included the definitions. This solves the issue of the type definitions not being present, but not the underlying issue of the

[3] Integrated Development Environment (IDE).

[4] The C standard does not provide precise definitions of the sizes of integer types, instead only requiring that each type be capable of storing a minimum range (BSI, 1999, pp. 21-22). Although exact-width types are part of the standard, they are optional (p. 255). A compiler will, because of this, generally rely on its internal type definitions to provide exact-width integers (such as the 8-bit `uint8_t` and 16-bit `uint16_t`), rather than using standard C types (such as `char`, `short`, `int`, and `long`, and `long long`).

conditional compilation symbols apparently being incorrectly evaluated.

At a later time, on compilation, the compiler began to report in `em_usbd.h` (the header providing USB device-related definitions) that a number of types defined by the driver could not be found. This was resolved by manually adding to the driver's source code an inclusion of the `em_usbtypes.h` header. It could not be determined why this error suddenly appeared, but it is believed that it is likely related to the above-discussed issue with incorrect evaluation.

General firmware architecture considerations

To be integrated with the proof-of-concept prototype firmware, the firmware for the USB prototype would (as noted in Appendix G1.2) need to be designed around the use of a periodic timer. While it would be possible to have the firmware work without being so designed, a design based around a periodic timer where the start of an operation is triggered by a timer pulse negates any concern relating to one interrupt firing before the interrupt service routine for another has finished.

While the driver includes an interface for firmware to use the driver's internal timer (which is a microcontroller timer specified in `usbconfig.h`), the interface is limited compared to direct use of a microcontroller timer. The driver-exposed timer has a resolution of 1 ms, and must be restarted each time it expires. These are not huge limitations, but must be taken into consideration.

In addition, the interface presented by the driver does not fit especially well with firmware designed around a periodic timer—events (such as receiving a packet from the USB host) are handled using callbacks, which are asynchronous with respect to the periodic timer and so would not fit into the program flow of any code handling the expiry of the periodic timer. It was initially considered that the callback could handle only what is required for basic operation, storing remaining state for code handling periodic timer expiry to retrieve and operate on, but this is not viable as the callback is required to report its status before it yields. This in mind, the next most practical method is likely to have the USB-interfacing code and any fan-controlling code operate independently, with any shared state (such as information about the current status of connected fans) protected by mutexes. As the integration of the two codebases is outwith the scope of the project (and likely not possible due to time constraints), these considerations minimally impact the development of firmware for the USB prototype.

Operating system-specific descriptor retrieval

As mentioned in Appendix G1.2, the Microsoft Windows operating supports two methods of retrieving its proprietary feature descriptors—version 1.0 or 2.0 of its Microsoft OS Descriptors specification. These methods are conceptually similar, but have important practical differences.

Each method uses a vendor-specific USB request, with the `wIndex` field set to indicate which descriptor (or set of descriptors) is to be retrieved. As there are a limited number of request codes, the request code for the request is specified in another descriptor which is retrieved first. This avoids any risk of a device which responds to a particular request code being harmed by an attempt to query for a proprietary descriptor. For version 1.0, the request code (`bMS_VendorCode`) is retrieved from a string descriptor at index `0xEE`. Microsoft has stated this caused devices which did not expect such a request to enter an indeterminate state, and so to require a reset before they would operate correctly (2017, p. 3). To avoid this issue, version 2.0 uses the Binary Device Object Store (BOS).

The BOS is a “framework for describing and adding device-level capabilities to the set of USB standard specifications” (USB-IF, 2013, § 9.6.2), and comprises a header descriptor and a set of device capability descriptors. The header indicates the total number of capabilities present, and each capability provides information specific to its purpose—for example, the `SUPERSPEED_USB` descriptor informs a host of whether a device supports operation at slower USB speeds. To enable use by third parties, there is defined a `PLATFORM` descriptor, which includes a UUID[5] specified by the third party (in this case, Microsoft) to indicate support for the party’s particular extension to the USB protocol. As Microsoft’s compatibility descriptor indicating that WinUSB should be used is part of version 2.0 of the Microsoft OS Descriptors specification, use of the BOS is required, and use of the BOS means that the device must support aspects of the USB 3.x specifications.

In particular, the device must be treated, for the purposes of the USB standard, as an “Enhanced SuperSpeed” device operating in a USB 2.0 mode. The result of this is that the device must indicate a USB version of 2.1 (`0x0210`) rather than a version of 2.0 (`0x0200`) and provide `USB 2.0 EXTENSION` and `SUPERSPEED_USB` descriptors in the BOS. As the microcontroller only supports operation in Low and Full Speed modes (Silicon Labs, 2014, p. 240), it appears that the Link Power Management protocol need not be implemented—the USB 3.1 specification states that “an Enhanced SuperSpeed device […] shall support LPM when operating in USB 2.0 High-Speed mode” (2013, § 9.6.2.1). It is necessary to confirm this by observing the operating system’s behaviour immediately after device connection.

To gather information on operating system behaviour, the device was configured to respond with the minimum required for enumeration and was connected to a host computer. The host’s behaviour was observed to be the following sequence:

1. The host requested the device descriptor.
2. The host requested all configuration descriptors.
3. The host requested string descriptor `0xEE`, which failed.
4. The host requested the device serial number.
5. The host requested the device’s list of supported languages.
6. The host requested the device qualifier descriptor, which failed.
7. The host declared enumeration complete.
8. The host requested the device product name.

This sequence is largely as would be expected, but provides confirmation—the host will not attempt to retrieve BOS descriptors for a USB version of 2.0, which means that a version of at least 2.1 must be reported. The failure to retrieve a device qualifier descriptor is no cause for concern, as that descriptor is optional

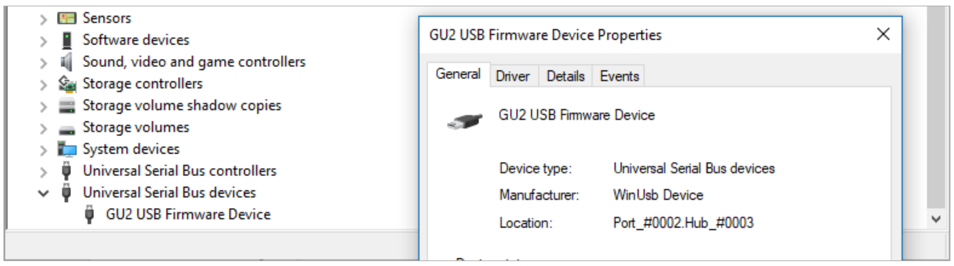
[5] Universally Unique Identifier (UUID), a 128-bit value generated in such a way as to make the likelihood of another person generating the same UUID extremely unlikely. A UUID is generally used where centralised registration is not desired (Leach, et al., 2005).

and included only if the device is operating at Full Speed but is capable of High-Speed operation (or vice versa). Similarly, the request for string `0xEE` is a request for the string descriptor indicating support for version 1.0 of the Microsoft OS Descriptors specification, and so its failure is not concerning.

The firmware was modified to report USB version 2.1, with other modifications to the reported descriptors as appropriate, and the test reperformed. As would be expected, the behaviour of the host was near-identical—the only difference in behaviour was that, after the host requested all configuration descriptors, it also requested the BOS descriptors. Before the protocol in Appendix D can be implemented, then, all that is required is for the appropriate `PLATFORM` capability be reported in the BOS, and the request specified in the Microsoft OS Descriptor specification be implemented.

The firmware was modified to report the Microsoft `PLATFORM` capability, and to include the Microsoft proprietary descriptor indicating WinUSB compatibility. A simple program for the host was written to locate the USB device and indicate success if the device was found. The device was connected to the host computer, the host computer's enumeration of the device observed, and the program started and confirmed to locate the device. In doing this, a bug in the Microsoft Message Analyser software may have been discovered—the device, when queried, would provide `USB 2.0 EXTENSION` and `SUPERSPEED_USB` capability descriptors, but the Message Analyser software would not indicate that the latter was present. The descriptor was confirmed present using USBView, a utility which is part of the Windows Driver Kit (WDK) and which lists the USB devices connected with any descriptors reported for them.

This issue aside, and using both the Message Analyser and USBView to confirm the presence of descriptors, an issue with the firmware was discovered. The BOS `PLATFORM` capability defined by Microsoft includes a 4-byte operating system identifier. The USB specification requires multibyte fields be transmitted in little-endian format (i.e. least-significant byte first), but the operating system identifier was being transmitted in big-endian order, resulting in the operating system not querying for the Microsoft OS Descriptors. This issue resolved, the device was connected and successfully registered as a WinUSB device (see below).


Sensors
Software devices
Sound, video and game controllers
Storage controllers
Storage volume shadow copies
Storage volumes
System devices
Universal Serial Bus controllers
Universal Serial Bus devices
GU2 USB Firmware Device
GU2 USB Firmware Device Properties
General
Driver
Details
Events

GU2 USB Firmware Device

Device type: Universal Serial Bus devices
Manufacturer: WinUsb Device
Location: Port_#0002.Hub_#0003

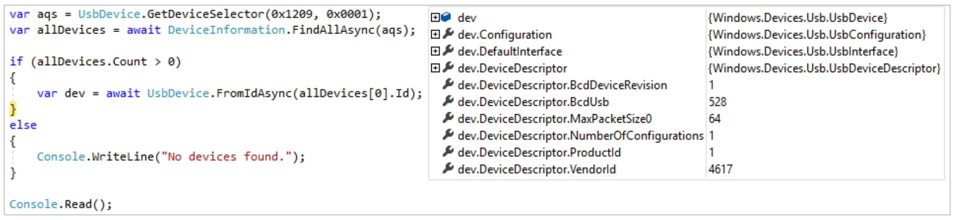
Before the simple program could be used as a test, it was necessary for the device to report a device interface GUID[6]—an identifier representing a programming interface exposed by a driver and consumed by other software, which enables that other software to interact with a device through its driver. For example, a device interface class may be used to provide an agnostic interface for multiple

[6] Globally Unique Identifier (GUID) is another name for a UUID. Documentation provided by Microsoft uses "GUID" rather than "UUID," and so it is also used here.

similar devices—Microsoft uses the example of three computer mice, connected by USB, serial port, and infrared port, respectively, and each exposed by the same device interface class (Microsoft Corporation, 2017).

The process for reporting a device interface GUID appears simple: in addition to the compatibility descriptor for WinUSB, the device reports a registry property descriptor with the name `DeviceInterfaceGUID` and value of that GUID. When the device is configured by the operating system, that descriptor is retrieved and the appropriate entry is added to the Windows registry. However, when added to the set of descriptors reported by the device, the operating system failed to recognise the device citing "an invalid Microsoft OS 2.0 descriptor set." To test whether the new descriptor was invalid, the WinUSB compatibility descriptor was removed from the descriptor set and the device reconnected. The operating system, without the compatibility descriptor, recognised the device and inserted the appropriate value into the registry. This indicates that no issue exists with the format of the registry property descriptor, but does not explain why the device is not recognised with both descriptors. As an experiment, the order in which the device sent the descriptors was switched—the registry property descriptor was sent first, and the WinUSB compatibility descriptor second—and, in this order, the operating system recognised and configured the device without error.

The simple program was then started and, as can be seen below, was successfully able to access the USB device.



```
var aqs = UsbDevice.GetDeviceSelector(0x1209, 0x0001);
var allDevices = await DeviceInformation.FindAllAsync(aqs);

if (allDevices.Count > 0)
{
    var dev = await UsbDevice.FromIdAsync(allDevices[0].Id);
}
else
{
    Console.WriteLine("No devices found.");
}

Console.Read();
```

```
dev                                          {Windows.Devices.Usb.UsbDevice}
dev.Configuration                            {Windows.Devices.Usb.UsbConfiguration}
dev.DefaultInterface                         {Windows.Devices.Usb.UsbInterface}
dev.DeviceDescriptor                         {Windows.Devices.Usb.UsbDeviceDescriptor}
dev.DeviceDescriptor.BcdDeviceRevision       1
dev.DeviceDescriptor.BcdUsb                  528
dev.DeviceDescriptor.MaxPacketSize0          64
dev.DeviceDescriptor.NumberOfConfigurations  1
dev.DeviceDescriptor.ProductId               1
dev.DeviceDescriptor.VendorId                4617
```

Accessing the USB device now possible, the next step is the implementation of the protocol set out in Appendix D.

Implementation of the Appendix D protocol

This section deals with implementation of the Appendix D protocol in firmware, and is mirrored by a section with the same title in Appendix G1.5.

The simplest portion of the Appendix D protocol to implement is the descriptor setting out the configuration of the fan controller—which simply lists how many fans the controller supports and the version number of the specification to which the controller conforms—and so is the obvious starting point for implementation. The descriptor is eight bytes long, and is to be included with the configuration descriptors returned by the USB device. For the sake of simplicity, the device will report that it supports the control of one fan, and that it implements protocol version 0.11. On the firmware side, this was simple—all that was needed was to add the appropriate bytes to the array containing the configuration descriptors, and to update the `wTotalLength` value reported. This second step was initially overlooked, resulting in the new descriptor not being registered, but could easily and quickly be corrected.

This completed, the remaining aspect of the Appendix D protocol to implement is the set of USB requests a fan controller must support. Three requests are set out in Appendix D—one to retrieve fan identification and status information, one to set the fan configuration, and one to retrieve that configuration. The first being the simplest request, it makes sense to implement it first. As the microcontroller will not be connected to real fans, spoofed values will be returned here. To aid in verifying correct operation of the software, the current value will be varied on each request. Excepting this aspect, the implementation is very simple, and would consist only of a request handler which responded with a pre-set array. On testing this code, the software reported currents of 35584, 35840, 36096, and 36352 mA. Inspecting the data sent in the USB response revealed that the bytes of the field containing the current were swapped—instead of `[0x8C, 0x00]` (140 with little-endian byte ordering), the bytes were `[0x00, 0x8C]` (little-endian 35840).

The source of this issue was identified as the code for varying the current reported between requests, which used a `uint16_t*` (pointer to a 16-bit unsigned integer) to access the two `uint8_ts` (unsigned 8-bit integers). As C integer literals are given in big-endian order but represented at the byte level (when compiling for the ARM Cortex-M4) in little-endian order, it is believed that some confusion occurred in writing the big-endian literal `0x8C00` (where `8C` is the first byte encountered reading left-to-right but is the second byte in terms of bit order) and reconciling this with writing a multibyte field as little endian in a byte array, where `8C` is both the first byte read and the first byte in terms of bit order. A swap in the byte order in the literal from `0x8C00` to `0x008C` corrected the issue.

With `GET_FAN_ID` implemented, the next request is `SET_FAN_MODE`. This request is used by the host to instruct the fan controller in its control of fans, either with direct operation (specifying that a fan is to be operated at a particular voltage or speed) or indirect operation (specifying a number of voltages or fan speeds with associated temperatures, where the fan is to be operated at a specified voltage or speed based on the current temperature). As there is no real fan to be controlled, the implementation in firmware here need only verify the correctness of the data and store it. While Appendix D requires a fan controller to retain its mode and the associated mode data, there is little to be gained in testing from implementing this aspect of the protocol, and so this requirement has not been met.

In the interest of brevity, and as the code for each mode would be generally the same, only support for `SET_FAN_MODE` requests with voltage data is implemented.

In this demonstrative prototype, the code for handling the request is only part of what would be required in a real application. The code performs a sanity check on the length and verifies that the provided data is voltage mode data (reporting an error if it is speed mode data). The data is then read into a buffer and its correctness verified. If the data is valid, the device reports success (and otherwise reports an error). Real-world code would, at this point, convert the mode data into any internal representation (such as one better enabling translation of temperature to a voltage than a simple linear search through an array), measure the current temperature, and configure fans as specified in the mode data.

To provide an indicator of success, the current-reporting code was modified such that, after configuration, the current reported by the device would cycle through each temperature value specified in the configuration data.

On testing the firmware, the software reported that fan configuration had failed, and the periodically-updating display of reported current stopped updating. The test was repeated with a breakpoint set at the start of the function responsible for handling `SET_FAN_MODE` requests, enabling the direct observation of the contents of the USB device's memory. Observing the device's memory, it became apparent that either the host was not sending the correct data, or that the device was not receiving the correct data—using the parameters `5V@10C` and `10V@20C`, the array containing the data read by the device contained decimal 160 as its first value, a number of zero bytes, then decimal 8. While the byte pair decimal 160:0 would be valid, its value is not what would be expected (a first byte of 20) and the values of subsequent bytes are neither valid (none containing a set END bit) nor what would be expected (being all zero). To narrow the source of the issue down, the software was stepped through and the contents of the buffer sent to the device inspected immediately before transmission. The buffer contained correct values, hexadecimal `14:6A:29:D4`, appearing to indicate that the issue was with the host.

To investigate further on the device, a breakpoint was placed inside the callback provided to the `USBD_Read` function and a trace of USB activity was started on the host. Reperforming the test, the cause of the issue was almost immediately clear—the trace reported no data for the transfer, which prompted a closer look at the request, revealing that the software was sending an `IN` (device-to-host) request rather than an `OUT` (host-to-device) request. This issue was corrected in the software and the test reperformed, confirming correct function.

All that remains after the `SET_FAN_MODE` request is implemented is to implement `GET_FAN_MODE` which, in this demonstrative prototype, consists only of a copy of the buffer into which `SET_FAN_MODE`'s data was read. In a real-world application, where a memory-constrained microcontroller might not have sufficient RAM to keep a redundant copy of the data, the handler for this request may convert an internal representation to the on-the-wire format required. The code to handle this request was implemented, and worked without issue.

The firmware portion of the USB prototype is completed.

G1.4 Software: overview

Host driver stack

As discussed above and in chapter 11 of the main body of the report, WinUSB is to be used as the driver stack on the host computer. WinUSB provides the kernel-mode portion of a device driver, exposing a generic USB communications API and leaving to the user-mode portion the implementation of a USB protocol. The user-mode portion is provided by the device hardware vendor.

WinUSB is exposed directly through the Windows API, the use of which would require either the use of the C programming language or an external binding in another programming language. While use in this way is certainly possible, the priority in this case is the rapid development of a suitable prototype. It would therefore be preferable to use a managed language with a managed abstraction over the WinUSB C API.

Fortunately, such an abstraction is provided by the Universal Windows Platform in its `Windows.Devices.Usb` namespace. This abstraction enables the use of the C# programming language which, being managed and with an extensive standard library, is likely to enable or at least aid the speedy development of a prototype.

On the software side, use of the abstraction (and, presumably, WinUSB) requires knowledge of information which can identify the device. It would appear, from the documentation provided by Microsoft, that the Vendor and Product IDs are sufficient. If this is not sufficient, the abstraction exposes a number of other possible methods of identification—by device class, subclass, and protocol, or by use of a device class GUID.

G1.5 Software: discussion and remarks

Implementation of the Appendix D protocol

This section deals with implementation of the Appendix D protocol in software, and is mirrored by a section with the same title in Appendix G1.3.

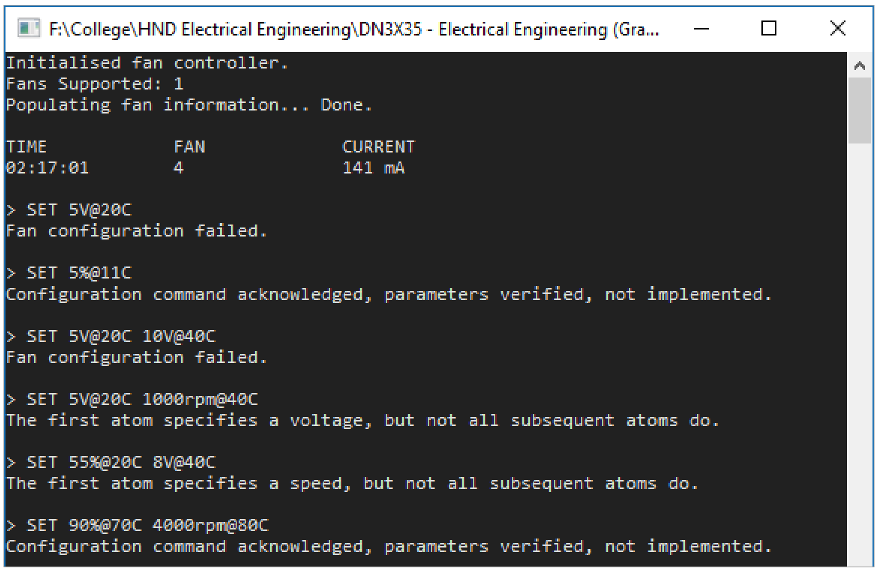
As discussed in that appendix, the obvious starting point for implementation is the reporting by the USB device of the fan controller configuration descriptor. As the `Windows.Devices.Usb` library exposes the raw descriptors reported by the USB device, all that is required is for the software to parse the descriptor, verify its contents, and assign the reported values as appropriate. There was a slight complication in that the library is intended for use with the Universal Windows Platform (and so did not use typical C# types), and so the use of a helper library with the necessary conversion functions was required, but otherwise no issues were encountered. Moving forward, interfacing with the controller will be via an abstracting `FanController` class, and the information retrieved from descriptors and queries (here, the information being the number of supported fans) will be exposed through properties and methods as appropriate.

As further discussed in that appendix, a sensible next step in implementation is implementing the `GET_FAN_ID` request. At a basic level, this is simple—a request using the appropriate class-specific request code is issued to the device, which returns eight bytes of information for the software to parse. This is complicated slightly by implementing this as part of an abstracting `FanController` class, and so instead this class maintains a collection containing status information for each fan. This collection is updated at the calling of an `Update` function by a user. Each time the user requests an update, the class issues a request for status information for each fan supported, parses this result, and updates the appropriate entry in the list of fan statuses. To display basic status information, the software calls the `Update` function in a loop, interprets and formats the updated information, and outputs the formatted information to the display. This aspect of the software was tested, and reported currents of 35584, 35840, 36096, and 36352 mA. Inspecting the data sent in the USB response, this was determined to be an error in firmware, and retesting after adjusting the firmware confirmed that the software correctly displayed the current reported by the device.

The next request in the Appendix D protocol is the `SET_FAN_MODE` request, used by the software to configure the fan controller and set how it is to control any fans connected to it. Accordingly, it is necessary for the software to provide some means for the user to specify desired settings, and so code was added to enable input processing. As the software continually updates its command line output, it is necessary for there to be a layer over the direct output which ensures that the status display remains consistent even while the software is reporting user input and the results of commands. The layer added in the software accomplishes this by moving the command line cursor to the appropriate position, writing the message provided by the function's caller, and returning the cursor to the position expected by the portion of the software responsible for displaying the fan status information. While the precise method used here would not be appropriate in real-world software, it is a method that is relatively simple and fast to write code for, and so is suitable for this task.

This layer was then used to define a simple command language consisting of a set of space-separated atoms, where the first atom represented the command (such as "`SET`" and "`GET`") and subsequent atoms any parameters to those commands. A parameter to the SET command consists of the setpoint (fan speed or fan voltage), followed by an "at" sign, and further followed by the temperature at which the controller is to manipulate the controlled quantity to the setpoint. For example, a parameter indicating 10 volts at 40 °C would be `10V@40C`, a parameter which indicates 95% speed at 80 °C would be `95%@80C`, and a parameter indicating a desired speed of 1300 rpm at 50 °C would be `1300rpm@50C`. However, as each mode would require generally similar code and for the sake of brevity, only the code required to implement voltage-based control is present.

The code is not complicated—the command processor reads in a command with a set of parameters, parses each parameter and categorises it based on the values it specifies, and verifies those values. The values are then passed to a method of the `FanController` class, which reperforms verification before serialising the points and sending a request to the USB device. The code was tested to ensure that the parsing and verification of parameters was correct (shown below).


F:\College\HND Electrical Engineering\DN3X35 - Electrical Engineering (Gra...

```
Initialised fan controller.
Fans Supported: 1
Populating fan information... Done.

TIME            FAN             CURRENT
02:17:01        4               141 mA

> SET 5V@20C
Fan configuration failed.

> SET 5%@11C
Configuration command acknowledged, parameters verified, not implemented.

> SET 5V@20C 10V@40C
Fan configuration failed.

> SET 5V@20C 1000rpm@40C
The first atom specifies a voltage, but not all subsequent atoms do.

> SET 55%@20C 8V@40C
The first atom specifies a speed, but not all subsequent atoms do.

> SET 90%@70C 4000rpm@80C
Configuration command acknowledged, parameters verified, not implemented.
```

The responses given indicate that the code is working correctly. The responses of "fan configuration failed" were expected, as the relevant firmware had not, at the time of testing, been implemented.

The commands shown in the image do not include a fan identifier. This would be present in real-world software, and would only require the addition of code to verify that the specified fan was one of those associated with the fan controller. It has been omitted here for brevity.

Once the relevant portion of the firmware had been written, the software and the firmware were tested together. As described in the discussion on the firmware, it became apparent that an error existed—the data received by the microcontroller bore no resemblance to that ostensibly sent by the software. Tracing a request, it was noticed that the request was specified as an `IN` (device-to-host) transfer and not an `OUT` (host-to-device) transfer. Reviewing the code for the software, this was as a result of the USB `SETUP` packet being constructed incorrectly, and the incorrect (but corresponding) method—`SendControlInTransferAsync`—being called to initiate the transfer. These issues corrected, the test was reperformed, and correct functioning of the software and firmware was confirmed.

With `SET_FAN_MODE` implemented, `GET_FAN_MODE` is the only request that remains to be implemented. As noted in the firmware discussion, the request is extremely simple (although the software is marginally more complicated than the firmware). On the software side, the implementation requests the mode and must deserialise it from the 2-byte entry format, checking the `END` bit for an indication of the end of entries. The deserialised data is then returned as a set of points (in the same way that a set of points was provided for `SET_FAN_MODE`) and these points displayed to the user.

The software portion of the USB prototype is completed.

G2 Sensors prototype

G2.1 Overview

A number of sensors are required for the fan controller to properly monitor and control the fans connected to it—the fan controller must measure the speed of a connected fan, the temperature of the environment, the voltage provided to the fan, and the current drawn by the fan. This appendix considers measurement only in the context of the proof-of-concept prototype, and deals with the practical portions of measurement rather than the relevant theory.

The proof-of-concept prototype firmware, which was tested in Appendix F and the code for which is contained in Appendix C3, includes an example of basic fan speed measurement, and so fan speed measurement need not be addressed here.

Temperature measurement

The proof-of-concept prototype includes a I^{2}C-connected NXP PCT2075 sensor capable of providing ±1 °C accuracy over a −25 °C to +100 °C range, making easy the process of obtaining temperature measurement—firmware would need only to configure the microcontroller I^{2}C peripheral, send the sensor an appropriate request, and convert the returned result into a usable format.

Voltage measurement

The voltage provided to a fan connected to the proof-of-concept prototype is monitored by a largely direct connection from the supply line to an input of an analogue-to-digital converter on the microcontroller, with the only components between the two a 3.83:1 potential divider which reduces the 12 volts provided to a fan to a level not exceeding 3.3 volts (a value falling within, and near the top end of, the rated range of GPIO input voltages). The only processing required would be the 3.83× multiplication of the ADC reading.

Current measurement

Despite current measurement being more involved than voltage measurement, it ultimately requires largely similar configuration of the microcontroller—a sense resistor is connected in series with the supply line, and an op-amp is connected across this resistance to amplify the voltage dropped across the resistance to a level suitable for measurement by a microcontroller ADC. In terms of processing, firmware would only be required to apply Ohm's law (compensating for the effect of amplification) and divide the measured voltage by the known resistance.

G2.2 Temperature sensing

I^{2}C initialisation

The vendor-provided emlib library includes functions for interfacing with the I^{2}C peripheral on the microcontroller, and so this can be used to communicate with the temperature sensor. The peripheral is configured for use using the `I2C_Init` function, but some prerequisite configuration is required before the peripheral can be configured or used.

Both I^{2}C peripherals are driven by the HFPER clock, and so must be enabled in the `CMU_HFPERCLKEN0` register as appropriate (either directly or through the emlib-provided function `CMU_ClockEnable`). Once this clock has been enabled, the peripheral's clock generator can be configured as required. This generator, as the microcontroller is the bus master, sets the clock rate for the I^2C bus, and cannot exceed the fastest speed supported by the slowest slave device—1 MHz in this case, as the temperature sensor is the only slave and supports "Fast-mode Plus" operation (NXP, 2017). The clock generator comprises a $^1/_1$ to $^1/_{255}$ prescaler and a control for varying the bus clock pulse width (with three predefined ratios of high periods to low periods, 4:4, 6:3, and 11:6, where the periods are the number of divided HFPER cycles).

The values used to configure the clock generator depend on the frequency of the driving clock (here, HFPER) and so vary from application to application. HFPER is itself driven by HFCLK, which (to enable use of the USB peripheral) will be operating at 48 MHz. The proof-of-concept prototype used a frequency of 14 MHz for HFPER, and so here HFPER is taken to be 12 MHz—the closest frequency attainable when dividing from 48 MHz with the division grades provided in the `CMU_HFPERCLKDIV` register. As there is no requirement for fast data transfer, an easily attainable bus frequency of 10 kHz is suitable. Applying equations 16.2 and 16.3 from the reference manual (Silicon Labs, 2014, p. 418), the configuration values to be used are a clock ratio of 4:4 and a $^1/_{174}$ prescaler. Using the emlib `I2C_Init` function, a caller can provide a ratio and frequency and have the library compute the appropriate configuration values.

With the I^2C clock generator configured, the next step in initialisation is to enable output from the peripheral to the microcontroller's pins. Enabling output is a two-part process—first, the relevant pins must be configured as wired-and[7] outputs and inputs, as appropriate, through the microcontroller's GPIO registers. After this, the peripheral output must be routed[8] and connected to the output channel for the routed-to output. For the proof-of-concept prototype, the I^2C peripheral in use is `I2C1`, which has its input and output on expansion header pins 7 and 9, which—according to the development kit manual—are connected to pins PC4 and PC5 on the microcontroller. These pins are specified in the EFM32WG990 datasheet as location 0 (Silicon Labs, 2013, p. 17; 2014, p. 66).

The GPIO configuration changes can be made using the emlib `GPIO_PinModeSet` function, or by manually setting the `GPIO_PC_MODEL[19:16]` and `[23:20]` fields to `WIREDAND` and `INPUT`, respectively. `WIREDANDPULL` and `INPUTPULL` modes are available (and would use microcontroller-internal pull-up resistors) but, in this case, external pull-up resistors are used, and so the internal pull-ups are not required and, if enabled, could cause incorrect operation.

Peripheral output routing is controlled by the `I2C1_ROUTE` register, and requires only that the location field `I2C1_ROUTE[10:8]` is set to location 0, and then that the `SCLPEN` and `SDAPEN` fields are set to 1.

I^2C communication

Emlib provides a pair of functions (`I2C_TransferInit` and `I2C_Transfer`) that

[7] Wired-and is the terminology used in the reference manual for an open-drain output, where the microcontroller would pull the signal to ground as required.

[8] EFM32 microcontrollers include reroutable IO, and so a signal from a single peripheral can be directed to up to 7 output pins, numbered 0–6 and called "locations."

enable managing an I^2C transfer. The first of these functions handles the initiation of a transfer, and takes as parameters the address of the I^2C device, a pair of data buffers, and a set of flags to modify the behaviour of the library. The I^2C address of the PCT2075 is `0x48` (`0b1001 000`), as the address is determined by the values of pins `A0` to `A2` and as those pins are, on the proof-of-concept prototype, tied to ground (NXP, 2017, p. 8). However, the documentation for emlib states that a 7-bit I^2C address must obey the `AAAA AAAX` format, where `A` is an address bit and `X` is a "don't care" bit. The value provided to emlib must therefore be `0x90` instead of `0x48`—equivalent to a bit-shift one position to the left. The remaining function parameters—the pair of buffers and the flags—must be considered together, as the flags affect how the library uses the buffers.

The flags describe four possible modes: reading data into the first buffer; writing data from the first buffer; writing data from the first buffer and then reading into the second buffer; and writing data from both buffers. The first two and the last mode correspond exactly to I^2C operations—an I^2C request consists of a 7- or 10-bit address plus a read/write (direction) bit, followed by data (from the master or from a slave, depending on the value of the direction bit), and so a write–read operation must consist of two requests. The last mode, a write–write operation, uses a single request, and may be to enable writing more bytes than one `uint16_t` can represent, as no length limitation exists with I^2C (NXP, 2014, pp. 10, 15).

The temperature sensor protocol is relatively simple—a number of registers are exposed, each assigned a 3-bit identifier, and each transaction with the device is an I^2C write to set the register in use followed either immediately by data to write to the register or by a separate I^2C read to retrieve the register value. Hence, the firmware would generally use the write and write–read modes in communicating with the sensor. To read the current temperature, the firmware would initiate a write–read transaction with the write buffer containing `0x00` and the read buffer being two bytes long.

Once the `I2C_TransferInit` function has been called, assuming that success is indicated by that function's return value, the firmware must wait for the transfer to complete. This could be significant time—with the microcontroller operating at 48 MHz and a 10 kHz bus frequency, the time taken to complete a write–read to retrieve the measured temperature would be at least 20 bus cycles, or around 2 milliseconds. In that time, 96,000 microcontroller cycles would elapse. As such, the firmware must repeatedly call the `I2C_Transfer` function to determine if the transfer is complete. On completion, the read buffer contains the data returned by the slave in response to the I^2C read.

Temperature sensor reading processing

The temperature sensor returns its reading as an 11-bit two's complement integer[9] contained in the most significant bits of a 16-bit big-endian response. The value of the reading is the temperature as a quantity of 0.125 °C increments, and so determining the temperature is a simple floating-point calculation: $0.125 \times N$, with

[9] Two's complement, as an operation, involves bitwise-inverting a value and adding one to the result. As a representation, two's complement uses the complement of a number as its negative representation—e.g. +3 is `0b011`, and so −3 is `~0b011+1 = 0b100+1 = 0b101`. This enables circuit simplification, as $A-B$ is equivalent to $A'+B$ and so subtraction can be done with an adder and inverter rather than a dedicated subtractor (Tan & Jiang, 2013, p. 412).

the value of N being the value returned by the sensor. Alternatively, if an even simpler operation is desired and as noted by the sensor datasheet (p. 12), it would only be necessary to read the most significant byte of the temperature register. This results in 1.00 °C resolution, obviating any need to perform a calculation.

G2.3 Voltage and current sensing

ADC initialisation

As with the I^{2}C peripheral used in temperature sensing, Silicon Labs' emlib library provides functions for interfacing with the microcontroller's analogue-to-digital converters (ADCs), including an `ADC_Init` function to configure the peripheral for use. Again, as with the I^{2}C peripheral, configuration of the driving clock and GPIOs is required before this initialisation function can be used.

The ADC is clocked by the High-Frequency Peripheral clock (HFPER), and so it is necessary to enable the ADC in the `CMU_HFPERCLKEN0` register. The ADC can only operate between 32 kHz and 13 MHz, and so it would be necessary to further divide HFPER using the prescaler in `ADC0_CTRL` if HFPER were operating at a higher frequency. Assuming, as in Appendix G2.2, that HFPER had been divided to 12 MHz, no further division would be necessary (unless, for example, reduced power consumption was desired).

As the ADC takes external input, each pin in use must be configured to act as an input. However, unlike other peripherals which can have their inputs and outputs routed to one of multiple possible locations, each channel of the ADC has a single location and the channel to use is selected instead. Configuration of a pin as an input is largely the same between peripherals—the relevant 4-bit field in the `GPIO_Px_MODEL/H` register for the particular port is set as necessary. In this case, `INPUT` is likely the correct mode—the low output voltage from the op-amps (likely in the low hundreds of millivolts) might not be sufficient to overcome the effect of a pull-down to ground. As the selection of a channel must be done during operation and not during configuration, it is not discussed here.

The ADC has variable resolution—it can both vary the number of discrete steps between 0 volts and its reference voltage V_{REF}, and the value represented by each step. Lowering the number of steps provides coarser measurement, and enables the ADC to measure at a higher rate; and adjusting V_{REF} enables higher-precision measurement at lower voltages (or differential voltages)—for example, an ADC with 8-bit resolution and a 5 V reference measures in 20 mV steps, while the same converter with a 1 V reference measures in 4 mV steps. Here, as speed is no great concern, use of the full 12-bit resolution is suitable. Selecting a voltage reference is more involved, and requires that the maximum reading be considered. Where the ADC is reading from a current transducer, the maximum value that must be read is (per chapter 14.3) not more than 1.65 V, meaning that the 2.5 V reference is suitable. When the reading is from a voltage transducer, the 3.83:1 divider ratio means that the maximum value is likely to be approximately 3.3 V—greater than any fixed reference provided by the controller. However, a reference of $2{\times}V_{DD}$ is permitted by the microcontroller, and would (for a 3.3 V supply voltage) provide a more than adequate resolution of around 1.6 mV. Use of the $2{\times}V_{DD}$ reference would require that the ADC occasionally sample V_{DD} for use in calculation.

Additionally, the ADC has a configurable oversampling mode. When enabled, the input signal is sampled multiple times and then filtered to produce the result. As both signals are expected to largely constant, and as the resolutions are 1.22 mA and 1.61 mV, it is not anticipated that any great advantage would be derived from oversampling. However, a level of oversampling could be used if desired.

The remaining configuration requires little consideration—the ADC requires an amount of time to "warm up," and can be kept warm to enable measurement at a higher rate. While energy conservation is a factor in producing the fan controller, shutting the ADC off between measurements is unlikely to save any significant energy, and so there is no reason to not have the ADC continually kept warm. To time the correct warm-up time, the ADC must be provided with the number of HFPER cycles corresponding to at least 1 μs (referred to in vendor documentation as the timebase). For a HFPER of 12 MHz, the timebase is 12 cycles (although the clock frequency is not completely accurate, and so may fall below the nominal frequency and may require that the timebase be adjusted upwards to account for periods where the frequency falls below the nominal).

This configuration done, the ADC can be enabled for use.

ADC operation

The ADC has two modes of operation—single sample mode, which generates a result from a single input; and scan, where the ADC iterates over a set of its inputs and produces a result for each one. The ADC can operate, irrespective of mode, in response to a trigger or continuously until stopped.

When operating in scan mode, the ADC samples the input, writes the result to the `ADCx_SCANDATA` register, and sets a bit in the `ADCx_STATUS` register. A result, if it is not read before the next result is written, is overwritten. This would initially appear to require that the processor continually poll the status register, removing any advantage gained from automatic sampling. However, the ADC can be used with the Direct Memory Access (DMA) controller[10] to autonomously copy each scan result to memory before it is overwritten, avoiding any need to poll a status register and simplifying code.

In the fan controller, the ADC is likely to be most useful in one-shot scan mode with DMA to transfer results to memory. For firmware designed around a master timer (such as the proof-of-concept prototype firmware), where each operation must complete before the end of a slice of time and so where polling the status register would not be viable (as there would be no guarantee that the check would occur in time to retrieve the result before it is overwritten), this configuration would enable better integration with the firmware. Further, as the ADC would be triggered in accordance with firmware state and not either continually sampling or triggered by an automatic timer, the consideration and code which would otherwise be necessary for handling (for example) a filled DMA buffer would not be required, as it could be ensured that the ADC is not triggered again before the buffer was read and its contents processed. This could result in simplified code.

To further simplify code, it would be possible to sample both current and voltage measurements in a single scan. Although this would require that current be taken

[10] DMA enables a peripheral to access the microcontroller's memory without intervention from the microcontroller's central processing unit (CPU), only involving the CPU when a transfer completes and data is available for processing.

using the $2{\times}V_{DD}$ reference rather than the 2.5 V reference, the resolution lost as a result of this change too insignificant to meaningfully effect measurement—with the 2.5 V reference there would be a 1.22 mA resolution, compared to a resolution of approximately 3.2 mA for $2{\times}V_{DD}$ if V_{DD} is taken to be 3.3 volts.

ADC reading processing: general

As discussed, the ADC provides a reading as a value between 0 and its configured resolution (here, 12 bits). Each step is equal to V_{REF} divided by $2^{RESOLUTION}$, and so some processing must be done to convert an ADC reading to an absolute value.

This processing is simple: the ADC reading is multiplied by V_{REF}, and then divided by $2^{RESOLUTION}$ to give the measurement in volts. Care must be taken to ensure that the calculation is performed using floating-point types, or that V_{REF} is multiplied by a suitable factor (such as ×1000 to give millivolts), so as to avoid inaccuracies from integer division.

ADC reading processing: voltage transducer

As the only conditioning applied to the voltage signal is division to reduce it to a safe measurement range, the measurement need only be multiplied by 3.83.

ADC reading processing: current transducer

The processing required to obtain a current value from the ADC measurement is only complex by comparison to the processing required for output from the voltage transducer. To obtain a current measurement, Ohm's law must be applied to find circuit current given a known resistance and the voltage dropped across that resistance, and the result must be divided to account for op-amp gain.

With a known resistance of 10 milliohms and op-amp gain of 50, the formula for computing current would be $^{V}/_{R} \times {}^{1}/_{G} = {}^{V}/_{RG}$. Conveniently, this can be simplified to a simple multiplication by a constant factor—10 milliohms is $^{1}/_{100}$ ohms, which becomes $^{50}/_{100}$ (or $^{1}/_{2}$) accounting for gain, and so the division is equivalent to multiplying the ADC measurement by 2. If the firmware was in assembly rather than C, this could be further simplified (for integer values) to a left bit-shift.

• END •

This report seeks to determine the theory behind the control of typical computer fans and apply this in the design and construction of a prototype for an electronic programmable fan controller, and to then apply the knowledge and observations gained in doing so to make recommendations for a design for a fan controller suitable for commercial and volume production.