

Version History

Version	Date	Changes
0.1	5 Feb 2018	Initial project specification and guide
0.2	6 Feb 2018	Table 5 pin numbers fixed and moved, Fig 11 enlarged
0.3	11 Feb 2018	Table 2 added PC7 pin line Table 2 fixed PC4 pin number on F334 to CN10-34 Table 6: updated TIC pin numbers to match Table 5 Added schematics for baseboard

1 Purpose of this Document

For 2018 the choice of processor, used in Design (E) 314, moves from the 16-bit Renesas RL78G13/G14 to a 32-bit ARM M4 processor: the ST Microelectronics **STM32F334R8**. The processor is supplied on a **NUCLEO-F334R8** development board.

The primary purpose of this document is to:

- Provide students with a clear **overview and scope** definition of the project;
- Provide clear **project requirements**, that will be used to test the hardware during demonstrations;
- Provide some assistance in understanding certain **concepts/information** about the components that will be used in the project;
- Identify the **critical design choices** that the student should solve.

2 Overview

The project will consist in building a hot water geyser controller/monitor system. The system has the following requirements:

- Per second measurements of the:
 - geyser supply voltage, to 5% accuracy
 - geyser supply current, to 5% accuracy
 - geyser water temperature, to 5% accuracy
 - ambient temperature, to 5% accuracy
 - geyser water flow rate, to 5% accuracy
- Per second control of the:
 - geyser element on/off state
 - water valve open/close state
- Scheduler/timer to heat water on predetermined times
- User interface

- Logging of system state, power usage, and water usage
- Debug/demonstration communication interface (specified in the Communication Protocol section)

Although the requirements of the system is given, they must be converted into measurable system specifications. Your system will be tested using automated demonstration stations, which will test whether your system meets these system specifications.

2.1 System block diagram

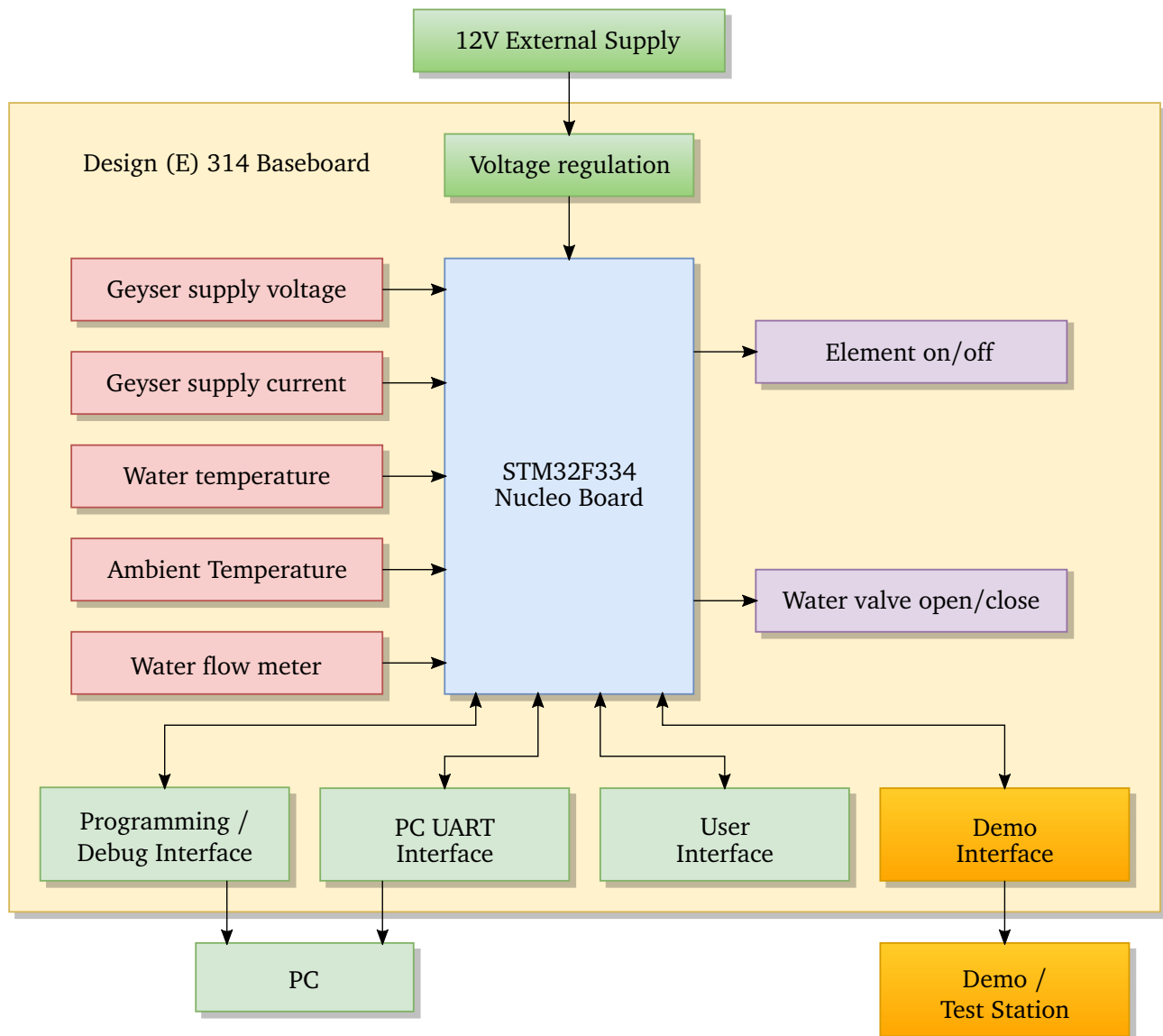


Figure 1: Block diagram for 2018 Design (E) 314 system

The system block diagram is shown in Figure 1, defining the main components of the system. As the module progresses more detail about each section will be discussed in the lecture sessions. However, this document should contain enough detail to enable you to finish the project, while your hardware and software meet the required specifications.

3 Hardware

The first change is that the processor module changes from a RL78G13 Promotion board to a STM Nucleo-64 development board. The basic development facilities provided are similar, both allowing debug of code though an on-board debugger module and most of the processor pins are brought out through headers.

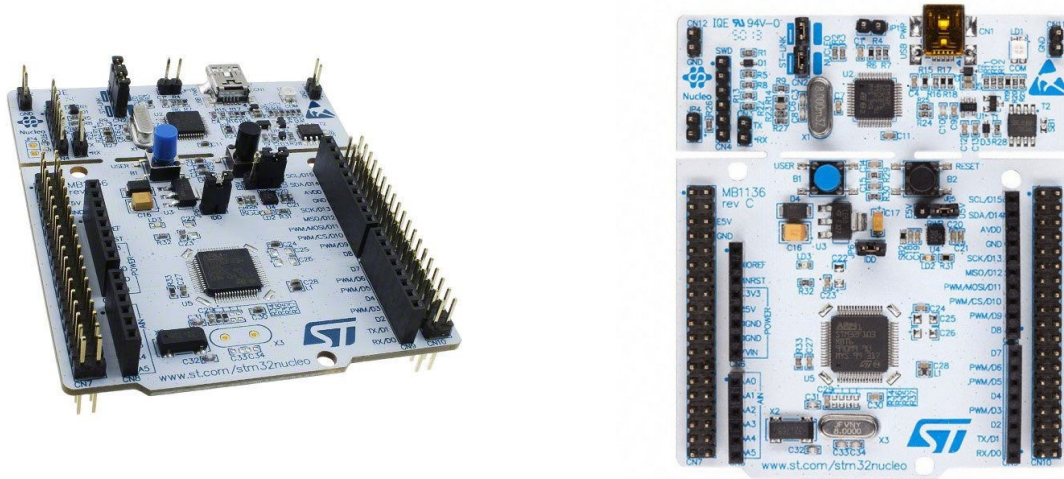



Figure 2: NUCLEO-F334R8 development board [4]

The new proposed baseboard layout with the STM Nucleo-64 module is shown in Figure ?? . Although the prototype area have shrunk compared to the previous board, the Nucleo-64 board allows stacking through either the Arduino headers or the Morpho connectors (on top).

To simplify the connection choices and possible solder problems, the following pre-determined design choices were made:

- The digital and analogue ground pins are hard-wired to the main board ground;
- The NUCLEO-STM32F334R8 is provided with main board power via the E5V pin (CN7-6) and F1 (fuse or link).
- No additional 3.3 V regulator is provided on the baseboard.
- To assist in building the system, some peripherals have pre-defined layout positions. These include:
 - 5V power regulation circuit;
 - Footprint and interface components for FTDI-UART module;
 - LCD module footprint;
 - Debug / user LEDs;
 - Debug / user push button switches;
 - Demonstration interface connector;
 - Miscellaneous connectors, eg. power, USB, screw terminal, audio.


i 3.3V power for prototyping is provided by the NUCLEO-STM32F334R8 LD3950 regulator via pin CN7-16. This should be adequate for up to 300 mA peripheral usage.

 The VIN pin should be left floating. To select board power, move the JP5 jumper on the NUCLEO-STM32F334R8 to the E5V (external power) position, from the U5V (USB power) position. If your board draws more than 100 mA do not try to power it using ONLY the USB power from the PC, as this could damage the USB port. It is safer to keep to jumper in the E5V position.

For the schematics and PCB layout of the 2018 baseboard, please refer to the *Baseboard2018.pdf* document on SUNLearn.

The majority of the Morpho connector pins are brought out to 5 connector areas (J4, J5, J9, J10, and J11) on the baseboard. Some pins are hard-wired, including power and ground. Given the specific layout and characteristics of the NUCLEO-STM32F334R8 and the baseboard in combination the following guidelines and information apply:


- The pins of the Arduino “analogue” connector CN8 on the NUCLEO-STM32F334R8 are brought out to solder pads J10 on the baseboard (odd and even numbered pins are linked).
- The pins on the lower “digital” connector CN9 on the NUCLEO-STM32F334R8 are brought out to solder pads J9 on the baseboard (again odd and even numbered pins are linked). Similarly the pins on the upper “digital” Arduino connector CN5 are brought out to J11 on the baseboard.
- The left-hand side of the Morpho connector CN7 on the NUCLEO-STM32F334R8 are brought out to J4 on the baseboard. The right-hand side of the Morpho connector CN10 on the NUCLEO-STM32F334R8 are brought out to J5 on the baseboard.
- The user pins, of the STM32F334 processor, on J4, J5, J9, J10 and J11 are fully numbered (in terms of their port number) on the baseboard.
- A small number of pins are also available from inside the NUCLEO-STM32F334R8 boundaries on the baseboard - PC9, PC11 and PD2.


 DO NOT CONNECT the following pins, of the NUCLEO-STM32F334R8, in your application unless you have a specific requirement - BOOT, 5V, RESET, VIN and any of the NC pins. The NUCLEO-STM32F334R8 operates normally without connections to these pins.

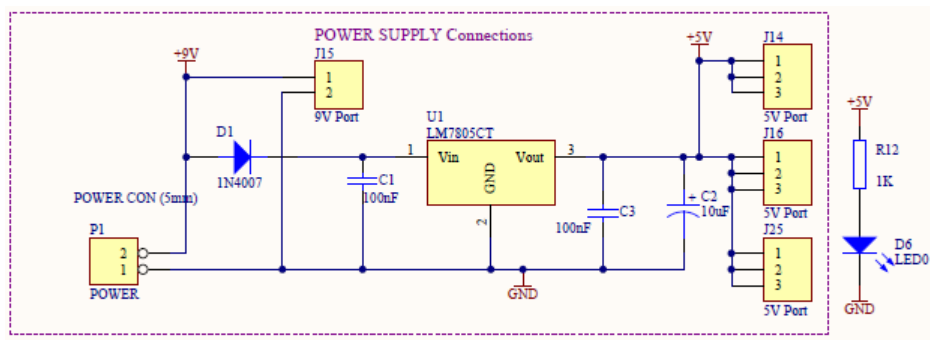
The following subsections will describe each subsystem in a bit more detail. Further detail regarding the required design steps will be supplied during lectures and, if needed, further documentation.

3.1 Power Supply

The power supply circuit on the main board uses a standard 7805 regulator (U1) in a TO220 package to regulate the input (9–15 V DC) down to 5 V. The 5 V power is routed to 3 jumpers (J14, J16 and J25, each providing 3 pins for wiring). An LED (D6) is provided to show that a voltage is present on the 5 V line.

 You will have to determine the minimum supply voltage input required as well as the maximum allowable input voltage. You must also make sure that the regulator (U1) is thermally stable (by adding thermal heat sinking hardware, if required). You must understand the role of each component in the power supply circuit.

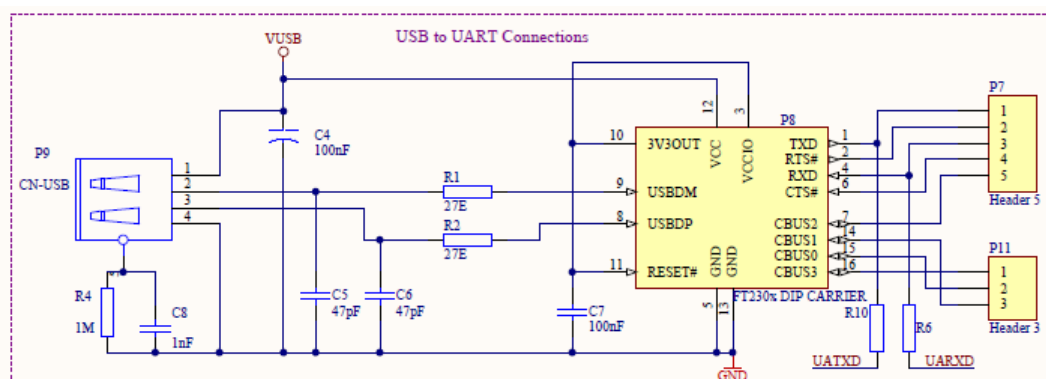
 Note that the 3.3 V supply is NOT present if the NUCLEO-STM32F334R8 is not attached!



3.2 USB to Serial Interface

An USB to UART interface, shown in Figure 4, is provided with the components P8, P9, R1, R2, R4, C4, C5, C6, C7 and C8. This interface relies on a USB type-B cable plugged into P9 and a FT230XS USB to Serial chip module mounted on P8. Only the TXD, RXD and GND lines are required to be connected for UART communications.

i You should use the given values for the passive components on your board. What is the role of R6 and R10 in the circuit? You will have to calculate sensible values for them.



This circuit uses power from the USB bus for communications. If connected correctly, the FT230XS device enumerates automatically and should present itself as a virtual COMM port.


i To test the circuit, make sure that the circuit enumerates correctly when the USB cable is connected to the PC. Secondly, a simple communications test can be performed by temporarily shorting the TXD and RDX pins (loop-back test), and checking that characters sent via a terminal program is echoed back.

3.3 LCD

The LCD Module will NOT form part of the 2018 project.


3.4 Switches


Space for five push-button switches are provided for simple menu, or other, functions. While the button is pressed, the button pins are shorted in pairs, otherwise they are open-circuit.

 *All button pins are floating to provide you complete freedom in connection choice.*

3.5 Debug / user LEDs

Space for four LED's plus series resistors are provided for display or debug purposes.

 *These pins are also floating on the base board - they are not connected to any other signal. You need to wire them to the pins/signals you wish to use.*

 *Remember to limit the current through the LEDs by using a series resistor; otherwise you will damage the processor output pins!*

3.6 NUCLEO-STM32F334R8 Connections

The NUCLEO-STM32F334R8 plugs into the baseboard through two separate 38-pin double row connectors, CN7 (left) and CN10 (right). These connections correspond to the NUCLEO Morpho connectors. The outer row connections are linked to solder connections J4 (for the odd numbered CN7 pins) and J5 (for the even numbered CN10 pins). The majority of the inner row CN7 (even numbered) pins that correspond to Arduino connections are brought out through J10, and the odd-numbered CN10 pins are brought out through J9 and J11.




 *It is good practice to trace the PCB connections on your board and familiarise yourself with the connections made via the PCB and therefore the interfaces available to you.*

Table 1 provides a cross-reference to the signals available on the Arduino, Morfo and Mainboard connectors. This table also contains some proposed and pre-determined signals to enable the use of UART and I2C communications. Other choices may be inhibited by the Azoteq ProxSense Cappel daughter board - more detail on this will follow later in the course.

 *The 5 V compatible pins are also listed — the rest of the pins are only 3.3 V compatible and need protection if connected to a 5 V source.*

 *You will need to refer to these pin connections whenever you need to decide on which pin to use for a specific function. Some peripherals have only one or two options, so where you have the design freedom, make sure you plan for future possible pin uses too.*

CPU Pin	Port	5V	Connector-Pin			Proposed Function
			Morpho	Arduino	Baseboard Solder Pad	
PA0	A	n	CN7-28	CN8-1	J10-1/2	UART2 TXD UART2 RXD LED2 SWD CLK SWD IO
PA1	A	n	CN7-30	CN8-2	J10-3/4	
PA2	A	n	CN10-35	CN9-1	J9-13/14	
PA3	A	n	CN10-37	CN9-2	J9-15/16	
PA4	A	n	CN7-32	CN8-3	J10-5/6	
PA5	A	n	CN10-11	CN5-6	J11-9/10	
PA6	A	n	CN10-13	CN5-5	J11-11/12	
PA7	A	n	CN10-15	CN5-4	J11-13/14	
PA8	A	Y	CN10-23	CN9-8	J9-1/2	
PA9	A	Y	CN10-21	CN5-1	J11-19/20	
PA10	A	Y	CN10-33	CN9-3	J9-11/12	
PA11	A	Y	CN10-14		J5-13/14	
PA12	A	Y	CN10-12		J5-11/12	
PA13	A	Y	CN7-13		J4-13/14	
PA14	A	Y	CN7-15		J4-15/16	
PA15	A	Y	CN7-17		J4-17/18	
PB0	B	n	CN7-34	CN8-4	J10-7/8	I2C SCL I2C SDA
PB1	B	n	CN10-24		J5-23/24	
PB2	B	n	CN10-22		J5-21/22	
PB3	B	n	CN10-31	CN9-4	J9-9/10	
PB4	B	Y	CN10-27	CN9-6	J9-5/6	
PB5	B	Y	CN10-29	CN9-5	J9-7/8	
PB6	B	Y	CN10-17	CN5-3	J11-15/16	
PB7	B	Y	CN7-21		J4-21/22	
PB8	B	Y	CN10-3	CN5-10	J11-1/2	
PB9	B	Y	CN10-5	CN5-9	J11-3/4	
PB10	B	n	CN10-25	CN9-7	J9-3/4	
PB11	B	n	CN10-18		J5-17/18	
PB12	B	n	CN10-16		J5-15/16	
PB13	B	n	CN10-30		J5-29/30	
PB14	B	n	CN10-28		J5-27/28	
PB15	B	n	CN10-26		J5-25/26	
PC0	C	n	CN7-38	CN8-6	J10-11/12	UART1 TXD UART1 RXD UART3 TXD UART3 RXD Blue Pushbutton 32 kHz Xtal 32 kHz Xtal
PC1	C	n	CN7-36	CN8-5	J10-9/10	
PC2	C	n	CN7-35		J9-13/14	
PC3	C	n	CN7-37		J9-15/16	
PC4	C	n	CN10-34		J5-33/34	
PC5	C	n	CN10-6		J5-5/6	
PC6	C	Y	CN10-4		J5-3/4	
PC7	C	Y	CN10-19	CN5-2	J11-17/18	
PC8	C	Y	CN10-2		J5-1/2	
PC9	C	Y	CN10-1		solder pad	
PC10	C	Y	CN7-1		J4-1/2	
PC11	C	Y	CN7-2		solder pad	
PC12	C	Y	CN7-3		J4-3/4	
PC13	C	n	CN7-23		J4-23/24	
PC14	C	n	CN7-25		J4-25/26	
PC15	C	n	CN7-27		J4-27/28	
PD2	D	Y	CN7-4		solder pad	
PF0	F	Y	CN7-29		J4-29/30	HE Clk 8 MHz
PF1	F	Y	CN7-31		J4-31/32	

Table 1: NUCLEO-STM32F334R8 to Baseboard connections

3.7 Azoteq ProxSense Cappel Daughter board


The Azoteq ProxSense Cappel daughter board interfaces with the Arduino connectors and provide the following features:

- A direct interface 4-digit 7-segment LED;
- A I2C bus interface capacitive linear- and rotary slider;
- A direct interface capacitive button input.

With this daughter board connected, some pins on the NUCLEO-STM32F334R8 are now used as shown in Table 2. Also shown is the other interface connections that are affected and their new logical choice.


CPU Pin	Port	5V	Connector-Pin			Proposed Function
			Morpho	Arduino	Solder Pad	
Cappo Reserved Connections						
PA0	A	n	CN7-28	CN8-1	J10-1/2	TOGGLE
PA1	A	n	CN7-30	CN8-2	J10-3/4	w_RDY
PA4	A	n	CN7-32	CN8-3	J10-5/6	s_RDY
PA5	A	n	CN10-11	CN5-6	J11-9/10	7_SEG_1 (LED2)
PA6	A	n	CN10-13	CN5-5	J11-11/12	7_SEG_2
PA7	A	n	CN10-15	CN5-4	J11-13/14	7_SEG_3
PA8	A	Y	CN10-23	CN9-8	J9-1/2	7_SEG_7
PA9	A	Y	CN10-21	CN5-1	J11-19/20	7_SEG_6
PB3	B	n	CN10-31	CN9-4	J9-9/10	7_SEG_11=Digit_4
PB4	B	Y	CN10-27	CN9-6	J9-5/6	7_SEG_9=Digit_2
PB5	B	Y	CN10-29	CN9-5	J9-7/8	7_SEG_10=Digit_3
PB6	B	Y	CN10-17	CN5-3	J11-15/16	7_SEG_4
PB8	B	Y	CN10-3	CN5-10	J11-1/2	I2C_SCL
PB9	B	Y	CN10-5	CN5-9	J11-3/4	I2C_SDA
PB10	B	n	CN10-25	CN9-7	J9-3/4	7_SEG_8=Digit_1
PC0	C	n	CN7-38	CN8-6	J10-11/12	I2C_SCL (wired)
PC1	C	n	CN7-36	CN8-5	J10-9/10	I2C_SDA (wired)
PC7	C	Y	CN10-19	CN5-2	J11-17/18	7_SEG_5


Table 2: Nucleo/Baseboard connections predefined for the Azoteq ProxSense Cappel Daughter board

 You should understand each of the pin functions given in Table 2 and the logical reasoning for their placement on the NUCLEO-STM32F334R8 connectors.


The two sliders are interfaced through two separate Azoteq IQS263 devices, both on a shared I2C bus, and the button through an IQS227 device.

The linear and rotary slider interface consist of a shared I2C bus for the two IQS263 devices, addressed as 0x44 for the linear slider and 0x45 for the rotary slider (7-bit address). Two separate open-collector active low ready lines, s_RDY for the linear slider and w_RDY for the rotary slider, complete the slider interface. These lines are bi-directional.

 In normal streaming or event mode the ready lines will be driven by the IQS263 periodically or when an event is signalled. The host may drive them low to force a communications window.

 The I2C bus for this board has moved to Arduino P4 pins 5 and 6, which on the NUCLEO-STM32F334R8 is CN8 pin 6 for SCL and CN8 pin 5 for SDA. On the baseboard this corresponds to J10 pins 11/12 for SCL and J10 pins 9/10. So for this board to communicate, you need to do the following modification:

- Wire SCL from J11-1/2 to J10-11/12 and SDA from J11-3/4 to J10-9/10 to form a wire link for the daughter board to move the I2C lines to the new positions.
- Set the PC4 and PC5 pins to input mode as not to interfere with the I2C communications.

 The IQS227 drives a TOGGLE line (active low, can be bidirectional as well), so an open collector interface is recommended here as well. This input will in this project only be driven by the IQS227.


The only other available UART1 connections are now on port pins PC4 and PC5.


The 7-segment display consist of 11 lines, with the bottom 7 driving the individual segments (active low) for all digits and the top four selecting the individual digits (also active low).

3.8 Voltage/Current/Temperature measurement


The geyser control application will require voltage, current and temperature measurements from the geyser system. In order to provide a safer interface, by not exposing students to 220V signals, the voltage, current and temperature readings from the geyser system will be emulated and provided as voltage signals via the Test-interface Connector, which is described in section 3.10.

To read these signals the Analogue-to-Digital Converter (ADC) peripheral of the STM32F344 controller will be used. The detail specifications of the analogue signals will be provided later in the course.

 Some of the analogue signals provided will have to be conditioned (translated to a sensible voltage level) and/or scaled, by circuitry on your you board, before they can be safely used as an input to the ADC of the controller. To get final values you will have to scale/manipulate the ADC sampled data in software.

 When working with an external signal always first check whether the voltage and current levels of the signal is within the acceptable safe limits of the pins you will connect the signal to. Use a multimeter, oscilloscope or other appropriate measurement instrument to verify signal levels.

The ambient temperature will be measured using an analogue temperature sensor, the MCP9700A [5]. More details on this topic will be covered later in the course.

 You will have to build support circuitry, and interface to the MCP9700A on the prototyping area of your baseboard.

3.9 Water flow meter

For simplicity's sake, the flow meter signal will also be emulated and provided via the Test-interface Connector. The signal will be a representative output of a practical circuit used to interface with the flow meter. A mechanical flow meter usually has a high/low digital pulse as output due to an internal contact

that opens and closes as the impeller turns. Each pulse edge represents a water flow unit. The faster the pulses the faster the water flow.

i Although the flow meter signal can be handled as a digital input, some signal conditioning will be required before you can safely connect the signal to your controller. Signal de-bouncing, either with circuitry or in software, will be required.

3.10 Test-interface Connector

A test-interface connector (TIC) is provided by P13. This connector is designed to be stackable and consists of an Amphenol Bergstik 16-pin surface mounted connector soldered to the bottom of the baseboard (solder side). This connector mates with a 16-pin Amphenol Dubox connector on the Test-Jig board (top side) that will be used during demonstrations. The connections provide a power source (9V), UART and 10 other connections during testing of the system.

This connector is supported by two solder connectors (J3 and J13) to connect to various ports or circuits.

i The CPU signals for the UART are assumed to be wired to J12. For your board, the jumper should be set to connect 1-2 and 5-6 during development. The jumpers should be removed during demonstrations and testing.

The jumper connector J12 simplifies the swap-over for the UART connections to and from the FT230XS USB to UART interface with jumper positions indicated in Table 3.

Mode	Jumpers	Note
S/W Development	1-2 and 5-6	FT230XS signals RXD and TXD are swapped over
Testing	Remove jumpers	Isolate FT230XS from P13 and CPU


Table 3: J12 Jumpers

More detail about the TIC, and it's complete pin descriptions are given in section 5.

4 Software Considerations

4.1 General

To develop software for this project, the same environment and typical development process as used in Computer Systems 245 will be used: Atollic TrueStudio with STM CubeMX. There is however one significant change: The use of a software version control system: GIT, described in more detail in section 4.2.


 In any of these project generator software set-ups, the user must be cautious. The software typically use comment delimiters to allow user code to be added, but this system is prone to deleting user code if the project is regenerated (adding new I/O for instance). Our advice is to add the minimum code in the project generated files and to use the project generator sparingly (ideally once only when generating the project for the first time). We recommend that you add files, with calls originating from the main C-file, which contain your user created code.

For a small project such as this, adding the following minimum files will simplify your development:

- Header file with all global definitions and function prototypes;
- Source file with all your global variables;
- Source file with initialisation function and user code, which leaves the main while loop short. Call a run function from main to execute your user code.
- Additional files to handle command decoding, LED display functions and communication are recommended.

4.2 Using GIT for Design E314

GIT is a form of source code version control. It not only keeps backup of your code, but also allows you to easily see changes in the code between checked-in versions, and facilitates collaboration on source code projects (although we will not use this latter functionality). Having knowledge of GIT and its processes is beneficial since it is used almost everywhere in industry where source code is part of the organization's Intellectual Property (IP).

 In this project you will be required to upload your code to a GIT source code repository, and share this repository with the lecturer(s).


4.2.1 Create a repository

For the project, you are required to create a user on `www.bitbucket.org`. If you already have a username on bitbucket, you may continue to use the existing user. After logging on to bitbucket, create a new repository, called `edesign_2018_<your student no>`. Use the settings as in figure 5.

We will use the bitbucket interface to place an initial file in the repository. After selecting the new repository, on the overview screen, select the 'Create a README' option. The content of the README file is not important and the student can populate this with his own information. See figure 6 for an example, and press 'Commit' button to finish.

You will also share your repository with the `sun_edesign_observer` user. This will allow lecturers to access your source code. To do this, after logging in to bitbucket website and selecting the repository you

Create a new repository [Import repository](#)

Owner  ▼

Repository name*

Access level ☒ This is a private repository

Version control system ☒ Git ☐ Mercurial

[Advanced settings](#)

[Create repository](#) [Cancel](#)

Figure 5: Bitbucket - New repository settings

Source

edesign_2018_12345678 /

Creating README.md

```

1 # README #
2
3 This is my source code for the 2018 EDesign314 project

```

Syntax mode: Markdown ▼ Indent mode: Tabs ▼ Line wrap: Off ▼ [Commit](#)

Figure 6: Bitbucket - Readme source example

created in the previous step, select the 'Settings' option on the left, followed by 'User and group access'. In the list of users, type 'sun_edesign_observer' and select 'Add' (see figure 7).

4.2.2 How to use the repository

In order to make use of the newly created bitbucket repository, we have to create, or 'clone' a local copy of it on the PC you are using for development. First we need to know the address (URL) of your bitbucket repository. You can get this from the overview screen of the repository on the bitbucket webpage, see figure 8, select the text next to HTTPS, and copy it to clipboard.

We will use an application called 'SourceTree' to manage our local copy of the repository. It provides a graphical user interface for most GIT tasks. After installing SourceTree (if required), select File -> Clone/New. In the window that appears, paste the repository URL, and supply a suitable location to place the repository files. See figure 9 and finish by pressing the 'Clone' button.

You can now use SourceTree to commit code to the repository, and synchronize local changes to the repository on bitbucket. There are numerous 'how-to' videos and guides available for GIT online. It is suggested

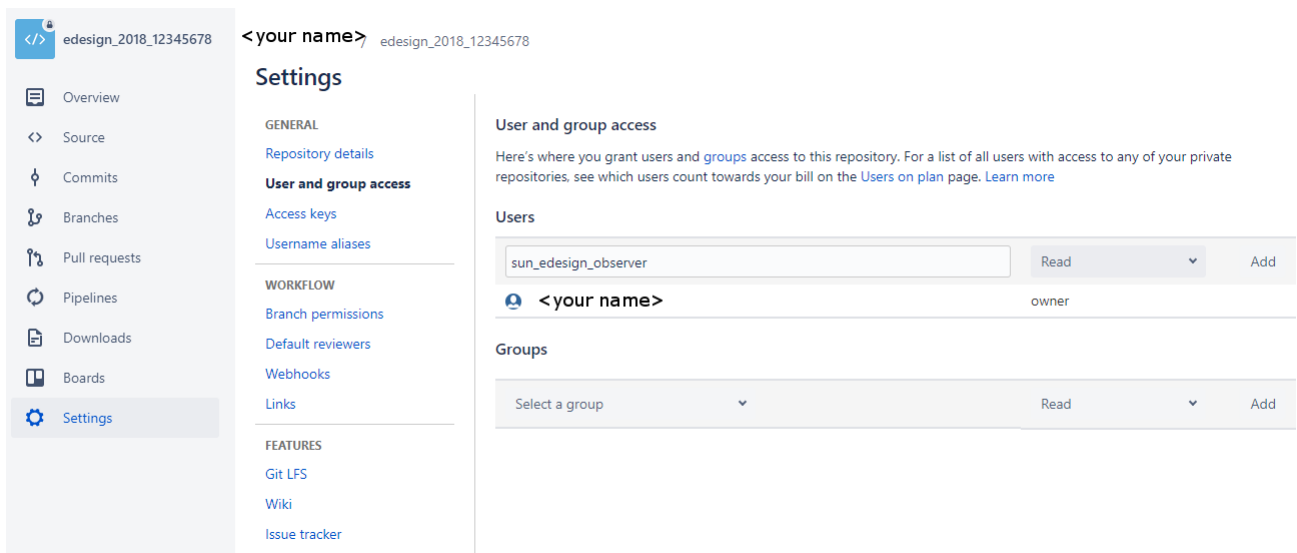


Figure 7: Bitbucket - Add user access

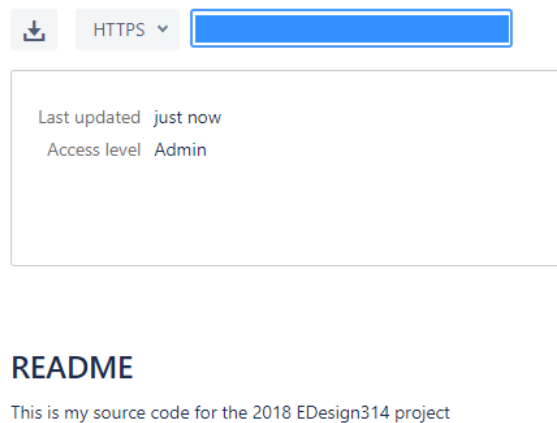



Figure 8: Bitbucket - Get URL

to start with the Atlassian introduction tutorial (Atlassian is the software developer of the SourceTree program): <https://confluence.atlassian.com/bitbucket/tutorial-learn-sourcetree-with-bitbucket-cloud-760120235.html>

4.2.3 Bitbucket repository and demos

You will be required to synchronize your latest source code prior to performing a demo. The source code in the bitbucket repository should reflect your code as used during the demo.

 *Marks will be deducted for failing to make use of bitbucket to make your source code available and work progress visible.*

4.3 Design of your System Structure

The next question the designer (you) will encounter is what software structure must be chosen. The suitable choices depend on the requirements at system level and the peripheral response times.

Lets take an example approach:

Figure 9: Bitbucket - Clone repository

The designer decides to put all the software in one big main loop with each function being executed sequentially before repeating indefinitely. What could go wrong?

Let us explore the possible problems by asking some questions about the software behaviour:

- How fast will the system respond to a UART command received? Will it consistently respond in this time?
- How fast will the system be able to sample digital inputs?
- How fast will the system be able to sample ADC inputs?
- How fast will the system be able to service the 7-segment LED outputs?
- How fast will the system respond to I2C inputs from the user?
- How long will the system take to do all the required calculations?
- How fast does the system NEED to do all the above, based on the specifications?
- Do all of these inputs/outputs require the same periodic attention from the system?
- How much resources will be consumed by the code? RAM, ROM and stack?
- Will the above answers change for one command versus another, versus many in quick order?

There is also a second, more subtle, reason for making a good software structure choice at the start of the project: How will any code additions or changes in sub functions affect the rest of the code? What you don't want is to have to rewrite a big part of your main loop (and maybe some other functions too), to accommodate a new function's requirements.

i *Modularity and well defined interfaces are key to keep the reworking of code to a minimum.*

In terms of the system design, we may attempt one of the following three approaches:

- The novice software writer will not consider a synchronous time-based structure for his code. Although it is feasible just to string all the functions together into one big while loop, the responsiveness of the system will be determined by the slowest function and recovery from time-outs and other errors are non-trivial to maintain.
- By using a timer-based schedule inside the while loop with interrupts can make the software much more robust. The choice of a tick-update period equal to $2-5\times$ the largest response delay time will simplify the code (otherwise a state machine and elapsed timer will also be required).
- The use of a real-time operating system (RTOS), such as FreeRTOS, will simplify the structure further, but add some overheads. It also has a steep learning curve. In the simple case of each thread having the same priority, the behaviour will resemble a large while loop (equal priority from interrupt but with pre-defined execution order).

The **large while loop** (second option above), reacting to interrupt flags, is the preferred option for this project and learning phase.

4.4 HAL vs. LL Functions

A programmer may choose to write code at register level, or may want the abstraction that the HAL (Hardware Abstraction Layer) provide. The CubeMX initialisation generator will generate code for either the HAL or the LL (Low Level) libraries, but not register-level code directly. By using the CubeMX generator, the student can only choose LL or HAL code (per peripheral), and with the documentation bias towards HAL code, it is likely that most users will start off with HAL code.

The majority of tasks for this type of project can be written using the HAL functions only. If required, direct register access is still available using the STM32f334 header file.

Interrupts are accessible by using calls with IT-ending (e.g. `HAL_UART_Receive_IT()`) and a default interrupt handler for every interrupt is generated (e.g. `USART1_IRQHandler()` in the `stm32f3xx_it.c` file). You can process the flags to determine the cause of the interrupt in this file, but the recommended call-back function (e.g. `HAL_UART_RxCpltCallback()`) is a better option.

The supplied HAL library has a number of drawbacks and/or bugs. There are significant bugs in the I2C interrupt handler — what we have identified are:

- No checking for zero condition in the transfer (which may cause additional characters to be transferred);
- The restart is conditioned by write and read direction requests. The approach works for alternating directions but does not work for repeated writes;

The effect is that the only call allowing repeated writes (utilising interrupts) fails during parameter uploading during initialisation but work fine during slider and event reading inside the main while loop while running.

It is therefore critical that students generate at least a new I2C write function capable of repeated starts for initialisation — a similar read function would be handy for checking parameters. An example is discussed in Appendix A.

A secondary effect is caused by the action that both the I2C and UART interrupt handler disables interrupts before returning — no problem if you are using deferred interrupt handling (setting a flag in the call-back and processing it in the main while loop) but back-to-back transfers through the call-back will not work.

4.5 ADC Conversions

The ADC is quite fast (worst case 2-3 microseconds conversion), so for undemanding applications, there is no need to really use either multiple channel conversion scan chains or even simultaneous sampling schemes. A simple single channel conversion using polling is sufficient.

The ADC can operate with as low as 1.5 clock cycle sampling for regular channels (if your input impedance is sufficiently low).

To convert a single channel, using the HAL library, we need to set up the channel, start the conversion, wait for completion and convert the result into the required scaled value.

4.6 Regular Interval Timing

The core ARM processor provides a SysTick timer as standard, which is set in the HAL library at 1 msec intervals, so there is no need to provide any additional timers for this simple purpose. A call-back from the SysTick timer will provide a 1 millisecond heartbeat.

```
void HAL_SYSTICK_Callback(void)
{
  msecCntr++;    // New millisecond timer
  msecFlag = 1;  // Flag to indicate millisecond event
}
```

4.7 UART

The format of the UART receive HAL functions are slightly at odds with how UART communication normally occurs, the HAL functions are geared towards known fixed length packets, while in reality we use variable length packets. To use the HAL receive function, we can set it up for single byte mode using interrupts (in effect to prime the receiver before the byte arrives), and as soon as we received a byte we can re-prime the receiver.

For UART transmission we can use standard block transmits.

4.8 I2C Communications

The I2C communications on the STM32 family is well supported by the HAL libraries, which remove most of the complexity from the user. The STM32F344 HAL libraries provide a variety of functions for reading from and writing to I2C devices, including calls that allow repeated start, interrupt and even DMA transfers.

Even with all the functions provided, not all the calls required are provided (no blocking-type of calls with repeated start) and there are bugs. The use of interrupt calls provide very little benefit in this application.

The writing and debugging of the functions will be daunting to new users, and the following strategy is proposed:

1. Start off with a single transfer — preferable a memory read (which is inherently an address write-repeated start-read transaction). The ideal memory location to read is the product identification register(s).
2. Develop an I2C write call (with repeated start, using the HAL blocking write call as basis), write to one of the configuration parameter registers and use the memory read listed above to read the written info back (to make sure the the write was successful).

3. Then develop an I2C read call (current address and with repeated start) and test as above.
4. Start to string commands together, and always make sure that you try to read the info back for checking purposes.
5. Use an oscilloscope, a low value on the SCL and/or SDA lines will quickly indicate a misplaced STOP token (it is more challenging to try and figure it out from the software only).

4.9 Real-Time Clock

The real-time clock should use the Nucleo 32 kHz oscillator (LSE) as clock source. The internal HSI main clock is not as accurate, so we would like to update the timing using the RTC clock as well. This can be done by enabling the wake-up timer in the RTC module and generating a 1 second wake-up interrupt. The call-back function used is

```
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef *hrtc)
{
    rtcSecFlag = 1;    // Flag to indicate 1 second period
}
```

This flag can now be used to reset the millisecond counter as well as maintain a parallel soft RTC. The problem with writing and reading from the RTC may incur two RTC clock cycle delays (about 60 microseconds), so we do not want to do this too often.

One aspect that needs to be remembered for the RTC is that both the time and date registers need to be read always as this completes the read cycle (not adhering will result in incorrect data), e.g.

The shadow registers are used by default, but knowing when to read can ease the delay — the best place is in the wake-up callback (as the seconds counter was also just updated — meaning no need for the shadow registers).

To write new time and calendar values, the RTC clock has to be stopped and init mode must be entered. Also note that the registers are write protected. The following HAL sequence may be used (there is no explicit HAL call to end the init mode — but it works).


```
...
// Update the Calendar (cancel write protection and enter init mode)
__HAL_RTC_WRITEPROTECTION_DISABLE(&hrtc); // Disable write protection
halStatus = RTC_EnterInitMode(&hrtc);      // Enter init mode
halStatus = HAL_RTC_SetTime(&hrtc, &rtcTime, RTC_FORMAT_BCD);
halStatus = HAL_RTC_SetDate(&hrtc, &rtcDate, RTC_FORMAT_BCD);
__HAL_RTC_WRITEPROTECTION_ENABLE(&hrtc);
...
```

4.9.1 Azoteq ProxSense Cappel Shield Board

The Azoteq ProxSense Cappel Shield Board provides a capacitive slider and button interaction with a 4-digit seven-segment LED display. The board provides a linear slider at the bottom, a rotary slider in the middle with a button at the centre.

I2C Slider Interface To set the TOGGLE, w_RDY and s_RDY correctly for open collector communications, make the set-up selection in the CubeMX project as output, open drain, with pull-up and high value. The line can be pulled low by writing a low value, and then again writing a high value will enable another master to control the line. The state of the line can be read at any time. Another option is to set interrupts for these lines if a millisecond driven while loop is not used. Use only the low transition (in particular for the TOGGLE line, as the others will go high after the I2C STOP token).

Seven-segment LED display To drive the seven-segment LED display, implement a segmented drive - i.e. update the display by driving each digit in turn for 5–10 mseconds, with the correct segment for that digit.

 You will have to convert raw data values to display to seven-segment characters and make sure to update the seven-segment unit at a rate fast enough to seem persistent to the user.

Azoteq IQS227 Button interface By default the IQS227 will pull the **TOGGLE** line low when the button is touched.

Azoteq IQS263 Communications The Azoteq IQS263 interface protocol is significantly more complex than the typical memory device. The following non-memory type of features are found:

- The register interface contains a number of registers, each containing FIFO type “memory” locations with varying parameter depth. This register structure implies that the data or parameters for any of the memory locations (and its parameters or data) can be addressed by a regular I2C memory call, but there is no way that the next address can be auto-incremented from here. To address the next register, a restart with new address is required.
- I2C communication is restricted to communications windows, driven by the IQS263 ready lines. If the device is set up correctly, these RDY cycles will happen at regular intervals in the streaming mode or when events are present in the event mode.
- The ready lines may be used by the STM32 microcontroller to request a communications window, but the actual communications is only possible when the IQS263 drives the ready line.
- When an I2C STOP token is received, the communications window is terminated and the read line released to high (and will initiate the start of a new IQS263 measurement cycle). This means that the all of the commands to set up the device, interrogate and change parameter values or read measurement data must be organized in blocks (and using restart tokens whenever the direction or address changes) with a single stop added at the end of the sequence.

The IQS263 can operate in either a streaming (default) or event mode. The nominal measurement cycle for the IQS263 is 40 Hz (25 msec) of which 5 msec is allocated for communications, by pulling the RDY-line low. The best solution is to set the device into the event mode (the IQS263 will only communicate if an event occurred) and then to lock unto the RDY line high-low transitions to initiate communications. After reading the data (one communications sequence), the RDY-line will transition high after the STOP token and the next measurement cycle starts.

The communication can be forced in event mode by the master pulling the RDY line low, wait 5–8 msec (to complete a measurement and/or sleep cycle), release the RDY line, and then wait for the for the IQS263 to pull it low and then communicate. This forced communication mode is useful to set up the device (and for handling stalled communications).

To set up the IQS263 initially, follow one of the two flow diagrams in #6.1 or #6.2 and the communications sequence in the figure on page 18 in [3].

The negative effect of using blocking calls during execution is keeping the processor active during the communications cycle (reading the events and delta timers from the IQS263. This can be verified using the oscilloscope, and/or using the loop statistics in Section 4.10. The results for a single slider channel with no other communication (which highlights the I2C communication loading) is shown in Table 4. The maximum timing value is the same as the total communication block width.

Comms baudrate <i>(kbaud)</i>	Measured Loop cycle time		
	<i>Maximum</i>	<i>Average</i>	<i>Units</i>
100	1772	76–90	μ sec
400	1191	55–65	μ sec

Table 4: Time penalty for using blocked I2C calls to a single slider

The positive effect of interrupt calls used in the main while loop will bring down the processing time considerably — an experiment at 400 kbaud showed that the maximum loop time was reduced to 26 μsec with the average loop time hovering around 8.5 μsec — the communications time is the same but the processor is not involved all the time and can do other tasks or sleep. There is the additional benefit of reduced power as well, but the run-time interrupt callback processing is significantly more complex and error-prone.

In the initialisation code, blocking code is simpler to use.

Slider position calculation The demonstration code suggest that the delta counters (8 bytes in total) are used as follows (ignore channel 0):

$$\sum_{\Delta} = \Delta_{c1} + \Delta_{c2} + \Delta_{c3}$$

$$p = p_{\max} \frac{\frac{1}{2} \Delta_{c2} + \Delta_{c3}}{\sum_{\Delta}}, \quad p_{\max} \leq 255$$

The value of \sum_{Δ} hovers around 150 when a finger is touching, 20-40 for a touch pen and drops to 0-2 for no touching. To eliminate glitches, limit the lower value of the sum or ignore the calculation when the sum is too low.

4.10 Main While Loop Statistics

Suppose we want to find out how long our loop times are for the main while loop. One way is to set up a timer (example of timer 7 here) with a prescaler that divides the 64 MHz clock frequency down to 1 microsecond intervals. The following code fragment illustrates how to do this:

```
while (1){
// Prepare timer 7 to get execution time statistics
__HAL_TIM_SET_COUNTER(&htim7, 0); // Reset the counter
HAL_TIM_Base_Start(&htim7); // Start Timer7 to get cycle time in usec


userProcess(); // Run the user process (your own cyclic program)

// Gather execution time statistics
HAL_TIM_Base_Stop(&htim7); // Stop Timer7
elapsedTime = __HAL_TIM_GET_COUNTER(&htim7);

if (elapsedTime > maxElapsedTime) // Update the maximum stats
maxElapsedTime = elapsedTime;
if (elapsedTime < minElapsedTime) // Update the minimum stats
minElapsedTime = elapsedTime;
// Average stats: Discrete IIR filter with 1/100 bandwidth
aveElapsedTime = 0.99 * aveElapsedTime + 0.01 * elapsedTime;

__WFI(); // Wait for the next interrupt
}
```

This code will provide minimum, maximum and average cycle times (to give you an indication of the processor loading). With the regular SysTick interrupt occurring every millisecond, the example of an average elapsed time of 20 microseconds indicate a 2% average loading.

 To use the timer in this mode, load the period value with a large value (such as 0xFFFF) as it will count up, if you leave it at the default of zero then the timer will not count!

5 Testing

Your student board (SB) will be tested during demo sessions, as specified in the study guide. The testing will occur in an automated fashion. Your SB will be plugged in to a test station (TS). The TS will provide power to your SB and generate a number of signals that are connected to your SB. Certain signals from your SB will also be monitored and checked by the TS.

5.1 Test Interface connector (TIC)

In order to facilitate testing, a test-interface connector (TIC) is provided by P13. This connector is designed to be stackable and consists of an Amphenol Bergstik 16-pin surface mounted connector soldered to the bottom of the baseboard (solder side). This connector mates with a 16-pin Amphenol Dubox connector on the Test-station board (top side) that will be used during demonstrations. The connections provide a power source (9 V), UART and 10 other connections during testing of the system.

Pin numbering of J13, J3 and P13 are as shown in Figure 10.

i From your board, you will have to route the signals listed in Table 5 to the test connector, P13, by making use of the solder connectors J3 and J13. The solder pads on J3 and J13 are routed to pins on the TIC (P13).

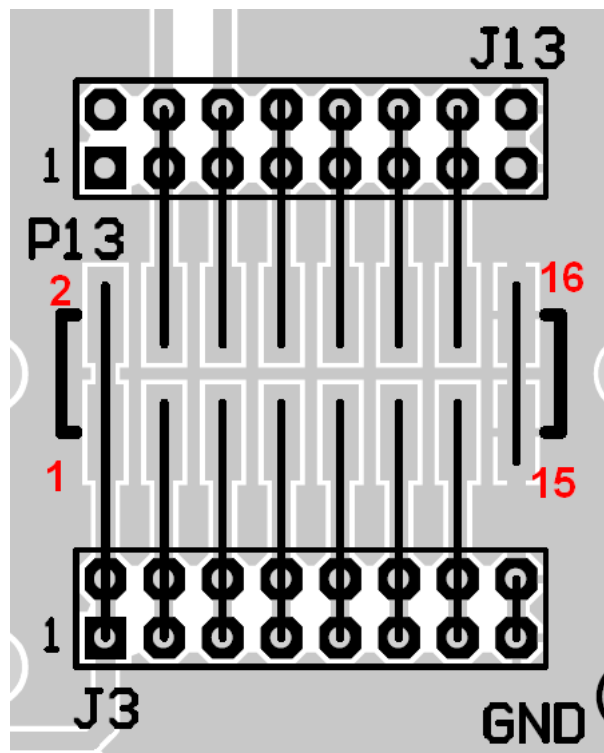



Figure 10: Pin numbers of J13, J3 and P13

P13 pin connection	J3/J13 solder connection	Signal	Direction
1	J3 - 1,2	V_bat - 9 V Supply from test station	SB <-> TS
2	N/C	V_bat - 9 V Supply from test station	SB <-> TS
3	J3 - 3,4	5V supply from student board (fed back to test station for verification/measurement)	SB -> TS
4	J13 - 3,4	UART transmit (TX from student board, RX of test station)	SB -> TS
5	J3 - 5,6	Student board temperature sensor output	SB -> TS
6	J13 - 5,6	UART receive (RX of student board, TX from test station)	SB <- TS
7	J3 - 7,8	Student board conditioned current measurement signal	SB -> TS
8	J13 - 7,8	Simulated heating element current - generated by test station (see section on Voltage/Current measurement)	SB <- TS
9	J3 - 9,10	Student board heating element control signal	SB -> TS
10	J13 - 9,10	Simulated heating element voltage - generated by test station (see section on Voltage/Current measurement)	SB <- TS
11	J3 - 11,12	Student board valve control signal	SB -> TS
12	J13 - 11,12	Simulated flow sensor measurement (see Water Flow meter)	SB <- TS
13	J3 - 13,14	N/C	N/C
14	J13 - 13,14	Simulated water temperature signal (See Temperature sensor)	SB <- TS
15	N/C	GND	SB <-> TS
16	N/C	GND	SB <-> TS

Table 5: Test-interface Connector Pin definitions

 Connections shown in grey in Table 5 are signals that are already routed on the PCB - you do not have to do anything to connect them.

5.2 Test method during demonstrations

The tests that the test station will execute once a student board has been plugged in, are designed to verify the requirements of the student board. These requirements, and the method in which the test station will perform the test are listed in Table 6

No	Requirement	Test(s)	
1	The student board shall be supplied from an external 9-10V DC source, and generate its own 5V supply for the microcontroller and peripherals. The generated 5V supply shall not vary by more than 5% (TBC).	Method	The test station will supply a DC voltage between 9 and 10 V to the student board on pin 1 of P13, and measure the voltage on pin 3 of P13.
		Pass/fail	The test passes if the voltage on pin 3 is within the specified tolerance of 5V
2	The student board shall make use of UART communications as specified in section TBD. If the student board receives a UART message that requires a response, such a reply shall be transmitted within 50ms of receiving the initial request.	Method	The test station will send a "request serial number" command to the student board using the UART link, and verify that a valid response message was received.
		Pass/fail	The test passes if the student board replies as per the UART specification in section TBD.
3	The student board shall measure current supplied to geyser element to 5% (TBC) accuracy.	Analog signal conditioning HW	
		Method	From the test station, apply a simulated Current Transformer output on pin 8 of the TIC. Through pin 7 of the TIC, the test station will measure the conditioned signal from the student PCB, and using the ADC of the test station MCU, determine if the signal was translated into the required ADC measuring range.
		Pass/fail	The test passes if the conditioned signal is translated according to section TBD, within 5% (TBC).

		Analog signal sampling	
		Method	From the test station, apply a simulated Current Transformer output on pin 8 of the TIC. Test station will request sampled current and voltage telemetry through the UART link, and determine if the simulated signal was correctly measured.
		Pass/fail	The test passes if the UART sampled current values corresponds with the signal that is being generated by the test station within 5% (TBC).
4	The student board shall measure geyser element supply voltage to 5% (TBC) accuracy.	Method	From the test station, apply a simulated heater element voltage output on pin 10 of the TIC. The test station will request sampled current and voltage telemetry through the UART link, and determine if the simulated signal was correctly measured.
		Pass/fail	The test passes if the UART sampled current values corresponds with the signal that is being generated by the test station within 5% (TBC).
5	The student board shall measure ambient (outside) and water temperature to 5% (TBC) accuracy	Ambient temperature sensor	
		Method	Through pin 5 of the TIC, the test station will measure the signal from the ambient temperature sensor on the student PCB, and using the ADC of the test station MCU, and an equivalent temperature sensor on the test station, determine if the student board temperature sensor is giving a valid output.
		Pass/fail	The test passes if the student board temperature sensor signal is within 5% of that of the equivalent temperature sensor.
		Water temperature sensing	
		Method	From the test station, an analog signal will be generated (on pin 14 of the TIC) to simulate the geyser hot water temperature sensor. The test station will request sampled temperature telemetry through the UART link, and determine if the simulated signal was correctly measured.
		Pass/fail	The test passes if the UART sampled temperature is within 5% (TBC) of the generated signal.
6	The student board shall measure water flow rate out of the geyser to 5% (TBC) accuracy.	Method	The test station will generate a series of pulses on pin 12 of the TIC to simulate flow from the water flow sensor. The test station will request sampled flow rate telemetry through the UART link, and determine if the simulated signal was correctly measured.
		Pass/fail	The test passes if the UART sampled flow is within 5% (TBC) of the generated signal.
7	The student board shall be capable of generating a TTL signal to control the switching of the heating element, when commanded to do so through the TIC UART interface. The signal that controls the heating element shall switch on within 50ms (TBC) of receiving the UART command.	Method	The test station will send a UART command to the student board to switch power to the element on and off. It will monitor pin 9 of the TIC to determine if the switching is implemented.
		Pass/fail	The test passes if the signal on P13 changes according to the command issued, within the time specified.

8	The student board shall be capable of generating a TTL signal to control the switching of the heating element based on an automatic schedule and temperature set-point	Method	The test station will issue a number of UART commands to set-up a heating schedule and temperature set-point on the student board. The test station will then monitor the signal on P13 of the TIC over time to see if the scheduled heating takes place. Throughout the test, the test station will also simulate a varying temperature sensor signal on pin 14 of the TIC, to verify that the heater switches off for temperatures above the set-point, and switches on for temperatures below the set-point.
		Pass/fail	The test will pass if all the following conditions are met: a. The heater is only switched on during valid times determined by the schedule. b. The heater is switched on if the sampled temperature falls below the set-point minus hysteresis value. c. The heater is switched off if the sampled temperature increases above the set-point plus hysteresis value.
9	The student board shall be capable of generating TTL signals to control a valve - to open or close the water supply to the geyser, when a command to do so has been received through the TIC UART interface. The signal that controls the valve shall switch on within 50ms (TBC) of receiving the UART command.	Method	The test station will send a UART command to the student board to open or close the valve. It will monitor pin 11 of the TIC to determine if the switching is implemented correctly.
		Pass/fail	The test passes if the signal on pin 11 changes according to the command issued, within the time specified.
10	The student board shall be capable of logging supply current, voltage, temperatures and flow rates, as well as the on/off state of the heater, at a 10s (TBC) sampling period, and storing the data in flash memory of the STM32F334R8 MCU.	Method	The test station will send a command to the student board to enable logging. The test station will generate a temperature, voltage, current and water flow profile, and output the simulated signals on pins 14, 12, 10, and 8 of the TIC. After some time, the test station will send a command to stop logging. The test station will request logged data through the UART link of the TIC, and verify that data matches the generated profiles.
		Pass/fail	The test will pass if the retrieved log data matches with the simulated data.
11	The student board shall display the current temperature on a 4-segment LCD	Method	The test station will generate an analog signal corresponding to a certain temperature on pin 14 of the TIC. The LCD will be inspected to see if the same temperature is displayed.
		Pass/fail	The test passes if the same temperature that was simulated from the test station is displayed on the LCD.
12	The student board shall allow a user to change the temperature set-point through a capacitive touch sensor interface	Method	The test station will instruct the user/demonstration assistant to adjust the temperature set-point to a specific value, using the touch sensor. The test station will read back the set-point using the UART link.

		Pass/fail	The test passes if the set-point that is read from the UART matches with the instructed set-point (to within 5% TBC).
--	--	-----------	---

Table 6: Test station test method definitions

5.3 UART communications interface

The UART shall operate at a baudrate of 115200 bps, with 8 data bits, 1 stop bit and no parity checking, using TTL signal levels.

UART messages make use of ASCII printable characters, to make it easy to interface with the student board debug connector using simple terminal programs, such as Teraterm and Realterm, in the absence of a test station.

Messages between the test station and the student board make use of a frame structure, where the message starts with a '\$' character, and ends with a two-character sequence - a carriage return character (0x0D) and line feed character (0x0A). In the text that follows the terminating sequence is displayed as <CR><LF>. This notation is only used to show the characters, but in reality the carriage return and line feed characters are not printable.

Communication parameters:

<i>Afrikaans</i>	Parameter	Value	Units/Notes
<i>Datateempo</i>	Data speed:	115200	baud
<i>Greepformaat</i>	Byte format:	8N1	no parity
<i>Beginsimbool</i>	Start delimiter:	\$	
<i>Veldskeiding</i>	Field separator:	,	
<i>Eindsimbool</i>	Terminating Delimiter:	<CR><LF>	Hex codes 0xD and 0xA

The student board shall act as a slave in UART communications. It will only respond to messages from the test station, and never transmit messages without a preceding request from the test station.

The data frame going from the test station to the student board shall have the following format:

Command frame, max 40 bytes							
Start byte	Cmd byte	,	n1 Bytes	,	n2 Bytes	..	<CR><LF>

And responses from the student board shall have the following format:

Response frame, max 40 bytes							
Start byte	Cmd byte	,	n1 Bytes	,	n2 Bytes	..	<CR><LF>

Response timing:

UART responses shall be issued with a MAXIMUM delay of 50ms (delay since the command was received by the student board).

5.3.1 Message formats

Message from test station			Reply from student board	Notes
Command character	Command/request	Data bytes	Data bytes	
'A'	Request serial number	None	Student serial number as ASCII string - 8 characters	

'B'	Open/close valve	A single character - '0' > close valve - '1' > open valve	Empty message	
'C'	Enable/disable automatic schedule	A single character - '0' > automatic heating disabled - '1' > automatic heating enabled	Empty message	
'D'	Switch heater on/off	A single character - '0' > heater off - '1' > heater on	Empty message	Only valid if the automatic heating schedule has been switched off
'E'	Enable/disable logging to flash memory	A single character - '0' > logging disabled - '1' > logging enabled	Empty message	
'F'	Set new temperature set-point	The new set-point value (integer between 0 and 100)	Empty message	
'G'	Request current temperature set-point	None	The current set-point value (integer between 0 and 100)	
'H'	Set time	HH,mm,ss Hours (24-hour notation), minutes and seconds	Empty message	
'I'	Get time	None	HH,mm,ss Hours (24-hour notation), minutes and seconds	
'J'	Set heating schedule	TBD	Empty message	
'K'	Request telemetry	None	I_rms, V_rms, ambient temperature, water temperature, flow rate, heater on/off	units and precise format TBC
'L'	Request log entry	Entry number (0 to 100)	TBD	

Table 7: Test message formats

In some cases, the student board will not return any data (commands 'B' and 'C' for instance). In this case, a message is still returned to the test station as an acknowledge that the original message was received, but the data bytes are omitted. In this case the reply from the student board will only contain the start-of-message '\$' character, the command ID from the original message and the terminating <CR><LF> sequence.

Examples

In order to retrieve the board serial number, the test station will send a command

\$A<CR><LF>

And the student board shall reply with (assuming the student number in this case is 12345678)

\$A,12345678<CR><LF>

In order to enable the automatic heating schedule, the test station shall issue a command

\$C, 1<CR><LF>

And the student board shall reply with

\$C<CR><LF>

In order to set a new temperature set-point the test station will issue a command

\$F, 60<CR><LF>

And the student board shall reply with

\$F<CR><LF>

In order to read the current temperature set-point from the student board, the test station shall issue a command

\$G<CR><LF>

And the student board shall reply with (assuming a set-point of 60 deg C)

\$G, 60<CR><LF>

5.4 Automatic heating schedule

The student board shall implement logic to automatically switch on the heater element based on a programmable schedule, and desired temperature set-point.

5.5 LCD and touchpad user interaction

The LCD shall normally display the currently measured water temperature in degrees Celsius. When the user interacts with the touchpad slider, the LCD shall display the current temperature set-point, and the slider will be used to change the set-point. When the user releases the slider, the LCD will revert to displaying the measured temperature.

A I2C Communications

A.1 General

The development of an I2C driver for a modern microcontroller such as the STM32F334 is not as complex as for the older microcontrollers such as the PIC16. The version on the STM32F334 is ST Microelectronics latest version (also found on the STM32F7 series and include a byte sequencer to help with the protocol. The critical information can be found in Chapter 27 of the User manual [1].

The HAL I2C routines provided work fine for simple transactions (single read, write or memory read and write). The memory read implements a write-restart-read sequence to read from a random address. The problem comes in using the available sequential functions (using interrupts) - the software tries to work for all versions and devices and cause hangs, e.g. bugs in the code do not handle the byte counters correctly for back-to-back transfers. There is no need to re-write all of the routines, single memory reads and writes will work properly for the simple reads and writes.

A.2 Low level — I2C Reads and Writes

At the lowest level we need an efficient read and write routine that can handle restarts (to switch transmission direction as well as single direction communication). The peculiar requirements dictated by the IQS263 forces batch communications in defined windows (i.e. the ability to string together a succession of reads and writes with only one stop at the end.

To achieve this, the suggestion is to use the software structure of the driver calls `HAL_I2C_Master_Transmit()` and `HAL_I2C_Master_Receive()`, but to revisit the flow and timing diagram in [1], figures 340, 343, 344 and 346 pages 901, 903, 905 and 907.

The code can be simplified and the Stop or RE-start case is handled using only the TXIS (transmit buffer), AF (Ack/Nack), TC (complete) and STOPF flags.

A.3 I2C Communication

It is normal to combine the low-level I2C read and write functions to create higher level communication sequences. Two standard uses are:

- When writing to a slave, the first data byte will typically be a register address on the slave, and then the data intended for that address will follow. The slave may (e.g. memory) or may not (e.g. IQS263) auto-increment the address to write to different registers.
- A random read require a write first with only the register address as data and restart, followed by a read to obtain the parameters. Again, auto-incrementing depend on the slave device. This will provide the same effect as the HAL memory read function, but with the additional freedom to control the stop bit.
- If interrupt code is used in the main while loop, only the first address write is triggered by the user code — the rest of the actions are driven by the I2C byte sequencer and interrupt call-back code. A state or phase counter is required to keep track of the read/write sequence in conjunction with with the interrupt call-back flags — in this application the runtime calls follow a write-read-write-read sequence to read the events and slider coordinates. This corresponds to a 5–6-state state-machine (four plus IDLE and/or final processing).

Expanding using this approach, function calls to simplify block reads, block writes and read-modify-write sequences can be generated.

Interrupt and Block code I2C call Comparison Let us compare the characteristics of blocking and various interrupt approaches:

Blocking calls An example of an I2C write blocking call would be:

```
...
halStatus = writeI2c(&hi2c1, Dev_Addr, ...);    // I2C write call
...
```

The processor will spin in a loop, responding to flags to process the call from start to end. The effect is a delay in operation, and the processor is using active power throughout the call.

Inline Interrupt calls If we now use an interrupt call, and wait for the transfer complete call-back, the code (using a hypothetical call `writeI2c_IT`) might look like

```
...
halStatus = writeI2c_IT(&hi2c1, Dev_Addr, ...);    // I2C write call
while(i2cTxFlag == 0){
;           // Wait for the setting of the transfer complete callback flag
}
i2cTxFlag = 0;
...
```

The code looks different from the blocking call, but the effect is exactly the same. The call return after starting the I2C byte sequencer, but we are still waiting for transfer completion.

Inline Interrupt calls, and sleeping If we now use an interrupt call, but go to sleep while waiting for the transfer complete call-back, the code might look like

```
...
halStatus = writeI2c_IT(&hi2c1, Dev_Addr, ...);    // I2C write call
do{
__WFI(); // Sleep while waiting for (any) interrupt
} while (i2cTxFlag == 0); // But wait for the right one!
i2cTxFlag = 0;
...
```

The effect is a similar delay, but with a significant power saving. Note that this code without the check (for the correct interrupt source) is not safe.

Interrupt calls, parallel call-back processing, and sleeping The code structure of the main while loop might look like

```
while(1){
...
halStatus = HAL_I2C_Master_Transmit_IT(&hi2c1, Dev_Addr, ...);    // I2C write call
...    // and carry on immediately with other tasks
...
if (i2cTxFlag){           // Now process the interrupt call-back
i2cTxFlag = 0;
...
}
...
__WFI(); // Finished processing, sleep while waiting for (any) interrupt
}
```

This code will achieve both fast response and low power.

References

- [1] *RM0364 Reference Manual, STM32F334xx advanced Arm[®]-based 32-bit MCUs*, ST Microelectronics, DocID025177, Rev 3, filename=en.DM00093941.pdf, September 2017.
- [2] *ProxSense[®] IQS263 Datasheet, 3 Channel Capacitive Touch and Proximity Controller with 8-bit Resolution Slider or Scroll Wheel*, Azoteq, IQS263 Datasheet V1.12, filename=iqs263_datasheet.pdf, September 2017.
- [3] *AZD088 IQS263 Communications Interface Guideline*, Azoteq, document V1, filename=azd088_iqs263_communication_interface_guideline.pdf, December 2015.
- [4] *NUCLEO-F334R8*, ST Microelectronics, <http://www.st.com/en/evaluation-tools/nucleo-f334r8.html>, accessed 2018-02-01.
- [5] *MCP9700A - Low-Power Linear Active Thermistor ICs*, Microchip, Rev G, filename=DS20001942G.pdf, <http://ww1.microchip.com/downloads/en/DeviceDoc/20001942G.pdf>, June 2016