4G Remote Monitoring Station

ARDUINO

If you have an expensive car, boat, caravan, holiday house, farm . . . virtually anything at all . . . you need to know what's going on when you are away. Is the battery going flat? Is your boat taking on water? Is your water pump running incessantly? You need to find out about these ASAP. All you need to do this is a couple of Arduino shields and a little software. You can even remotely trigger actions, such as switching off that misbehaving pump, before it drains all your water!

We have to admit: the raison d'être for this project originally had nothing to do with monitoring expensive cars or boats, remote holiday houses, farm water tanks or anything so esoteric.

It was all to do with wombats.

For the benefit of our overseas readers wombats, a somewhat threatened species, are cute, (usually) slow-moving furry animals that inhabit the Australian bush (and, incidentally, are unique in that their poo is cube-shaped!).

But even that's not the whole story.

Silicon Chip's zany resident cartoonist, Brendan Akhurst, actually lives way out in the bush and is a member of his local wombat protection society.

Part of their remit is to re-home wombats in areas where they are less likely to be attacked by other animals (eg, dogs). They do this by trapping them and relocating them.

The problem is/was that wombats are very easily stressed and will die if they are trapped for too long.

What Brendan wanted was a means of letting society members know, as soon as possible, that one of their traps had been sprung.

"Aha!" we said. "There is an idea we've been working on for a couple of months which will alert you, via your mobile phone, of virtually any incident."

"A sprung wombat trap included?" he asked.

"We did say virtually any incident!"

So Brendan's Wonderful Wombat Warning Whatchamacallit is the result. . .

Of course, what *you* use it for is entirely up to you!

by Tim Blythman

2G, 3G and 4G

The 2G (GSM) mobile network has already been essentially shut down, and some telcos are starting to threaten to shut down their 3G network.

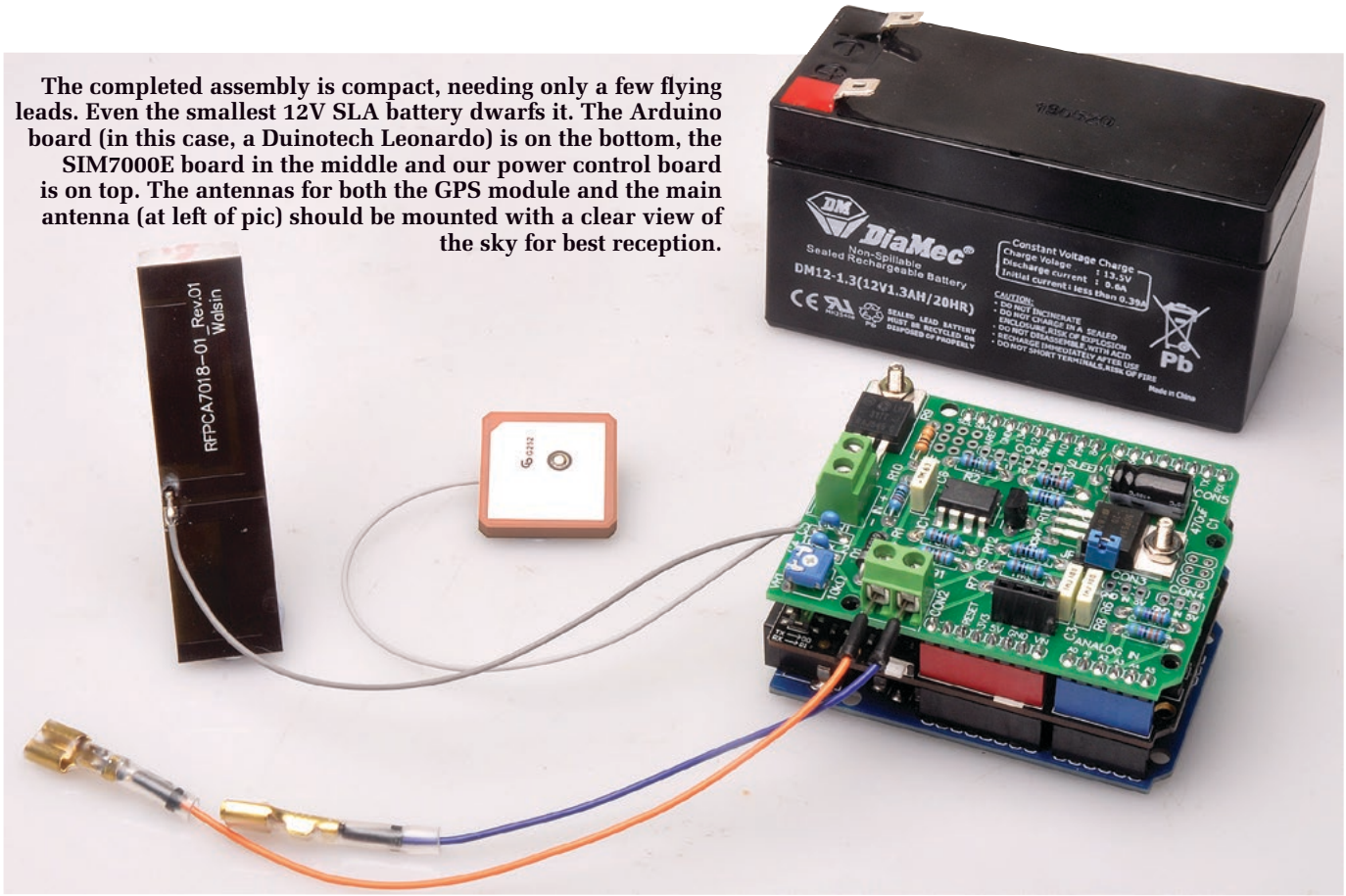
So to do this kind of job reliably for the next few years at least, you need a 4G device.

When we found a locally-available Arduino 4G shield at a reasonable price, we jumped at the opportunity to design a Remote Monitoring Station around it.

Since this Station is based on an Arduino board, it can be easily programmed to suit your specific requirements. It can monitor the state of switches, voltages, sensors – just about anything, provided there is an Arduino library to interface with it (and there usually is)!

Similarly, you can send commands to the Arduino from your mobile phone or PC to do things like switch mains power on or off, using a simple add-on device, such as our Opto-Isolated Mains Relay (October 2018; siliconchip.com.au/Article/11267).

You might remember the GSM Remote Monitoring Station project from the March 2014 issue, which was also



The completed assembly is compact, needing only a few flying leads. Even the smallest 12V SLA battery dwarfs it. The Arduino board (in this case, a Duinotech Leonardo) is on the bottom, the SIM7000E board in the middle and our power control board is on top. The antennas for both the GPS module and the main antenna (at left of pic) should be mounted with a clear view of the sky for best reception.

Arduino-based (**siliconchip.com.au/Article/6743**). That is now well and truly obsolete.

If you have already built that design, as the command set of the new 4G shield is very similar, you may be able to update it by simply replacing the shield and making some small code changes.

One minor difference between that 2G shield and the 4G shield we are using here is that the power control signal is on a different Arduino pin. We haven't tested this newer shield with the older Monitoring Station design, but it's likely to work with some fiddling.

The new shield has enough extra features to warrant a major update, and so this new 4G Remote Monitoring Station makes good use of many new features.

As this is an Arduino-based project, you'll need to be familiar with the Arduino IDE (integrated development environment) software.

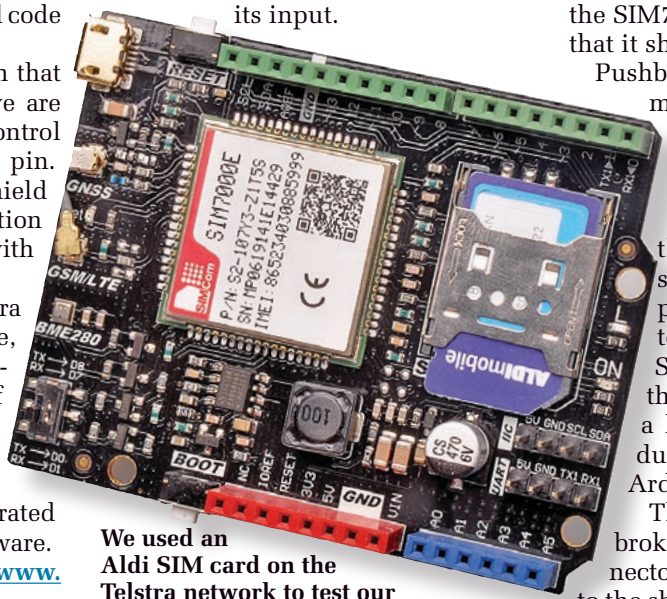
This is a free download from **www.siliconchip.com.au/link/aatq**

4G Shield

This project revolves around a 4G Shield designed by DFRobot. It is based on a SIMCom SIM7000E module, which provides the 4G capability.

Its circuit diagram is shown in Fig.1.

The SIM7000E module is fed power from the Arduino's VIN pin via an MP2307 buck regulator. This produces a steady 3.3V with at least 4.75V at its input.



We used an Aldi SIM card on the Telstra network to test our SIM7000 shield. This SIM cost $5 and did not need to add any extra credit, even after two months of testing. The shield also has sockets for external mobile and GNSS antennas.

While this eliminates the option of powering the shield from a 3.7V lithium-ion or LiPo cell, it will work with most Arduino applications powered from the VIN pin or DC barrel socket.

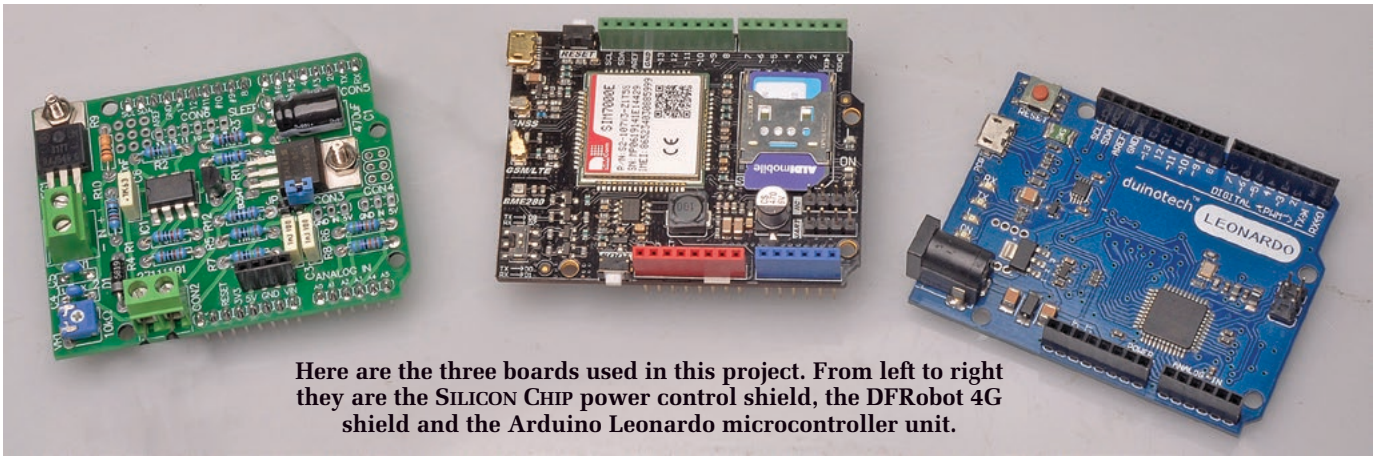
Below the regulator is the power control section. The PWRKEY pin on the SIM7000E is pulled low to signal that it should power on or off.

Pushbutton S1 connects this pin momentarily to ground, while NPN transistor Q1 allows Arduino pin D12 (driven high) to achieve the same effect.

Communication between the host Arduino and the shield is with a serial TX/RX pair, via level-shifting transistors Q8 and Q9. Slide switch S2 routes the signals to either D0/D1 (which is usually a hardware serial port on Arduino boards) or D8/D7 on the Arduino.

The SIM7000E's USB port is broken out to a micro-USB connector. This does not supply power to the shield, but can be used by a PC to communicate with the SIM7000E module.

We didn't investigate this in detail, but it appears that many features of the



Here are the three boards used in this project. From left to right they are the Silicon Chip power control shield, the DFRobot 4G shield and the Arduino Leonardo microcontroller unit.

module are useable via the USB connection. It may even be able to act as a USB 4G modem.

There are also sockets for a full-sized SIM card, 4G antenna and GNSS (Global Navigation Satellite System) antenna, which is used for GPS and GLONASS.

For more information on these and the many other GNSS systems that exist, see our article *"A look at SatNav systems"* in the November 2019 issue (siliconchip.com.au/Article/12075).

One common use for a remote monitoring station is vehicle tracking, and in this case, a GNSS receiver is practically mandatory.

We don't need to add any extra hardware to implement tracking into our 4G Remote Monitoring Station.

The two antennas are included when you purchase the shield. The mobile network antenna is a simple, self-adhesive PCB type.

Some photos of the shield show a small whip-style antenna, but it appears this has been replaced by the PCB type. A ceramic patch antenna is supplied for GNSS use.

Also on the shield is a BME280 temperature, pressure and humidity sensor.

We covered modules built from similar sensors back in 2017 (siliconchip.com.au/Article/10909). This is a great addition as it adds even more sensor data to our 4G Remote Monitoring Station without needing extra hardware.

We found that the temperature read by the sensor was higher than ambient, probably due to the heat generated by the surrounding circuitry; think of how hot some mobile phones get!

SIM7000E module

The SIM7000E module is touted as an NB-IoT/LTE/GPRS/GPS module. LTE and GPRS are longstanding mobile data transmission technologies, but NB-IoT is a newer standard.

NB-IoT is a low-power, narrowband variant of mobile phone technology, designed to be used by IoT (internet of things) devices. You can find out more about IoT from our November 2016 article on the topic (siliconchip.com.au/Article/10425).

We aren't using the NB-IoT feature in this project; at this stage, it appears the technology is still being rolled out in Australia, and an NB-IoT-specific SIM card is required.

The SIM7000 module comes in several variants which support different mobile frequency bands. We are using the SIM7000E (the model sold by Core Electronics), which is designed for the European market and supports bands 3, 8, 20 and 28. There is also a SIM7000C which is designed for the frequencies used in China.

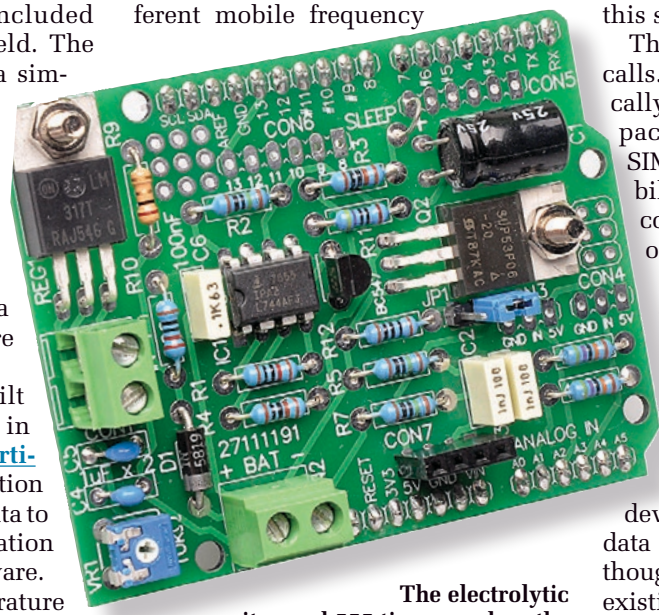
In our tests in suburban Sydney, we could not get reception with an Optus SIM card, but had success with a Telstra SIM card. This is despite the Optus network apparently using some of the above bands.

Because not all frequencies are offered in all areas, your experience may be different.

We suggest that you thoroughly research what frequencies are used where you plan to deploy the 4G Remote Monitoring Station, to make sure this shield supports them.

This module does not support voice calls. Most monitoring stations typically use SMS (text messages) or data packets for communication. The SIM7000E module does support mobile data, and this is a great way to communicate lots of small snippets of monitoring data.

Our design uses both the SMS and mobile data features of the shield.



The electrolytic capacitor and 555 timer used on the power control shield have been carefully chosen for low leakage and low quiescent current, to extend battery life. Note the jumper wire connecting the Arduino's D7 pin and the SLEEP terminal.

ThingSpeak data logging

Our Water Tank Level Meter from February 2018 (siliconchip.com.au/Article/10963) is a remote device which periodically uploaded data to the ThingSpeak website, although it used a WiFi connection to an existing internet-connected network, limiting where it could be used.

Such restrictions can be removed by using a 4G shield like this one.

We're also using ThingSpeak for this project. It has a simple API (application programming interface)

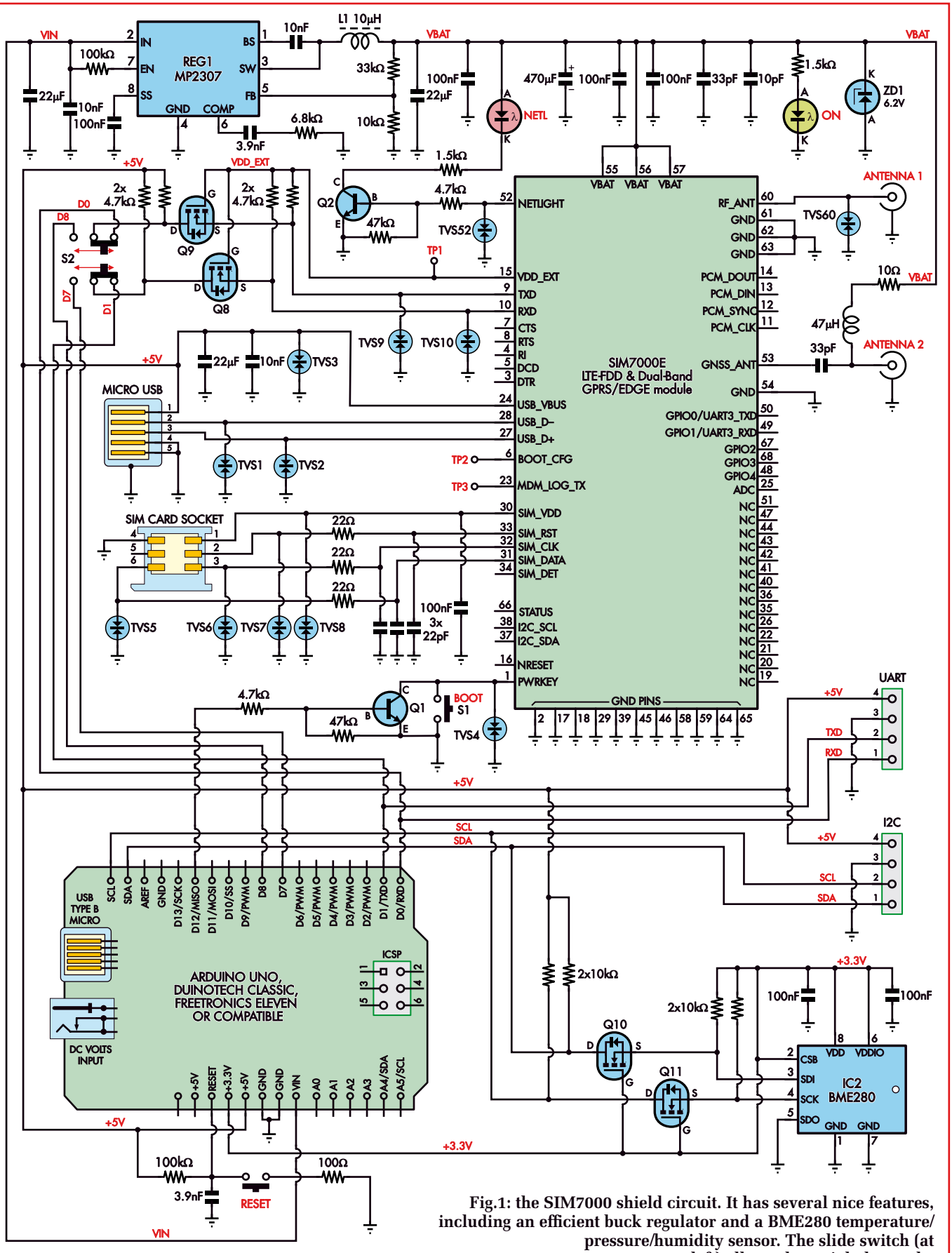


Fig.1: the SIM7000 shield circuit. It has several nice features, including an efficient buck regulator and a BME280 temperature/pressure/humidity sensor. The slide switch (at upper left) allows the serial data to be rerouted; we are using pins D0 & D1.

SIM7000 ARDUINO EXPANSION SHIELD

for uploading data, which is great for resource-constrained devices like Arduino microcontrollers. It also provides simple graphical visualisations of the recorded data. The data can also be downloaded as a CSV (comma-separated value) file.

These files can be opened into a spreadsheet program to allow more advanced analysis to take place. Creating charts is also an option in many spreadsheet programs.

Uploading data to the ThingSpeak website requires mobile data, so the SIM card used needs to support this.

For the low cost, longer-expiry prepaid mobile phone plan that we tried, it was typically cheaper to send weeks of data to the ThingSpeak website than to send a single text message.

Our 4G Remote Monitoring Station is ideally suited to providing continuous logging of data via 4G as well as sending text messages for raising alerts for unusual situations that need to be acted on promptly.

Power control shield

In addition to the pre-build 4G shield, our Remote Monitoring Station also uses a custom-designed shield to provide for battery power, solar charging of that battery and some power-saving techniques to give it a long runtime when using a small battery.

Most Arduino boards have poor power efficiency; they have simply not been designed with this in mind. Even with the processor set to sleep mode, other components such as linear voltage regulators and LEDs have quiescent currents in the tens of milliamps.

Our shield reduces the standby battery draw to microamps, which it does by completely disconnecting the Arduino board (and SIM7000 shield) from the battery using a Mosfet, and only powering those components up periodically.

The shield provides a reasonably efficient way to charge the battery, and also monitors the supply and battery voltages via the Arduino's analog inputs.

Most of the unused pins are broken out to headers, allowing other sensors or peripherals to be connected. There's even a small prototyping area on it, for extra components.

Shield circuit

The custom shield circuit is shown in Fig.2. Unregulated DC power is fed into CON1 (from a solar panel, plugpack etc), while the battery is connected via CON2.

The battery needs to operate in the range of 7-15V, so a 12V lead-acid or SLA battery is suitable. We used a 1.3Ah SLA with our prototype.

Power from CON1 feeds REG1, an LM317 adjustable regulator. The 220Ω fixed resistor and 10kΩ variable resistor VR1 allow you to set its output voltage.

As REG1 maintains about 1.25V between its OUT and ADJ pins, around 5mA flows through the 220Ω fixed resistor.

This current mostly also flows through VR1, so by adjusting its resistance, you change the voltage between ADJ and GND.

Hence, you can set the voltage at VOUT, since this will be the voltage across VR1 plus the 1.25V.

The output from the regulator is filtered by a 1µF capacitor and fed to the battery via 1A schottky diode D1. This prevents the battery discharging into the power source, eg, if it is a solar panel in darkness.

The 1Ω resistor between the output of REG1 and anode of D1 reduces the output voltage as the current drawn from REG1 increases.

Hypothetically, if the current through this resistor reached 1.25A (which would not be possible in practice), the voltage across this resistor would rise to 1.25V, cancelling out REG1's reference voltage, so the output would drop to 0V.

Thus, the output voltage drops approximately 1V for every 100mA of load current.

So if a battery is heavily discharged and its terminal voltage is low, the regulator output current is moderated until its voltage rises to the normal range, at which point virtually no current will flow into the battery.

In practice, the charging current is limited by dissipation to about 160mA for a 12V solar cell (with nominal 18V open-circuit voltage) feeding into a discharged 12V battery.

While the range of VR1 allows a terminal voltage from 1.25V up to 56V to be set, it shouldn't be set any higher than around 15V as this may damage the regulator on some Arduino boards, as well as IC1.

If you don't need to use a battery, power can instead be fed directly into CON2. D1 will prevent back-feeding into the charge circuit.

Arduino and SIM7000 shield power control

Power control is provided by 7555 CMOS timer IC1 and P-channel Mosfet Q2. Q2 is simply used as a high-side switch. Q2 can handle much more current than is required (hundreds of milliamps at most), so it does not need heatsinking.

We have chosen the CMOS variant of the 555 for its low quiescent current of around 60µA, compared to about 10mA for the bipolar version. This is because it is active and drawing current from the battery at all times.

IC1 is configured as a monostable timer. When power is first applied, the 470µF timing capacitor is discharged, and the threshold pin (pin 6) is below 2/3 of the supply voltage. The trigger pin (pin 2) is held high by the 10kΩ resistor. The transient conditions during power-up result in output pin 3 being high, and discharge pin 7 is in a high-impedance state.

With output pin 3 high, Q2's gate is high and so it is off, and the Arduino is not powered. The 470µF capacitor slowly charges up through the 1MΩ resistor. This capacitor needs to be a low-leakage type; otherwise, the leakage would prevent it from charging up fully.

The time constant of this circuit is nominally 470 seconds (just under eight minutes). Due to the 555's tripping point not being exactly 63% of the supply voltage, it actually takes around 10 minutes for the timer's state to change.

Once the trigger pin voltage reaches about 2/3 of the supply voltage, output pin 3 goes low, pulling down Q2's gate, switching it on and connecting the battery to the Arduino board's VIN pin.

This powers on the Arduino board and attached 4G shield. IC1's discharge pin, pin 7, goes low at the same time, discharging the 470µF capacitor quickly via the 470Ω resistor.

Being a monostable circuit, it remains in this state until the Arduino decides that it needs to power down.

To do this, it drives the base of NPN transistor Q1 positive, pulling the trigger pin (pin 2) of IC1 low. IC1's flip-flop toggles, output pin 3 goes high (switching off Q2 and the Arduino) and the discharge pin (pin 7) goes back to a high-impedance state, allowing

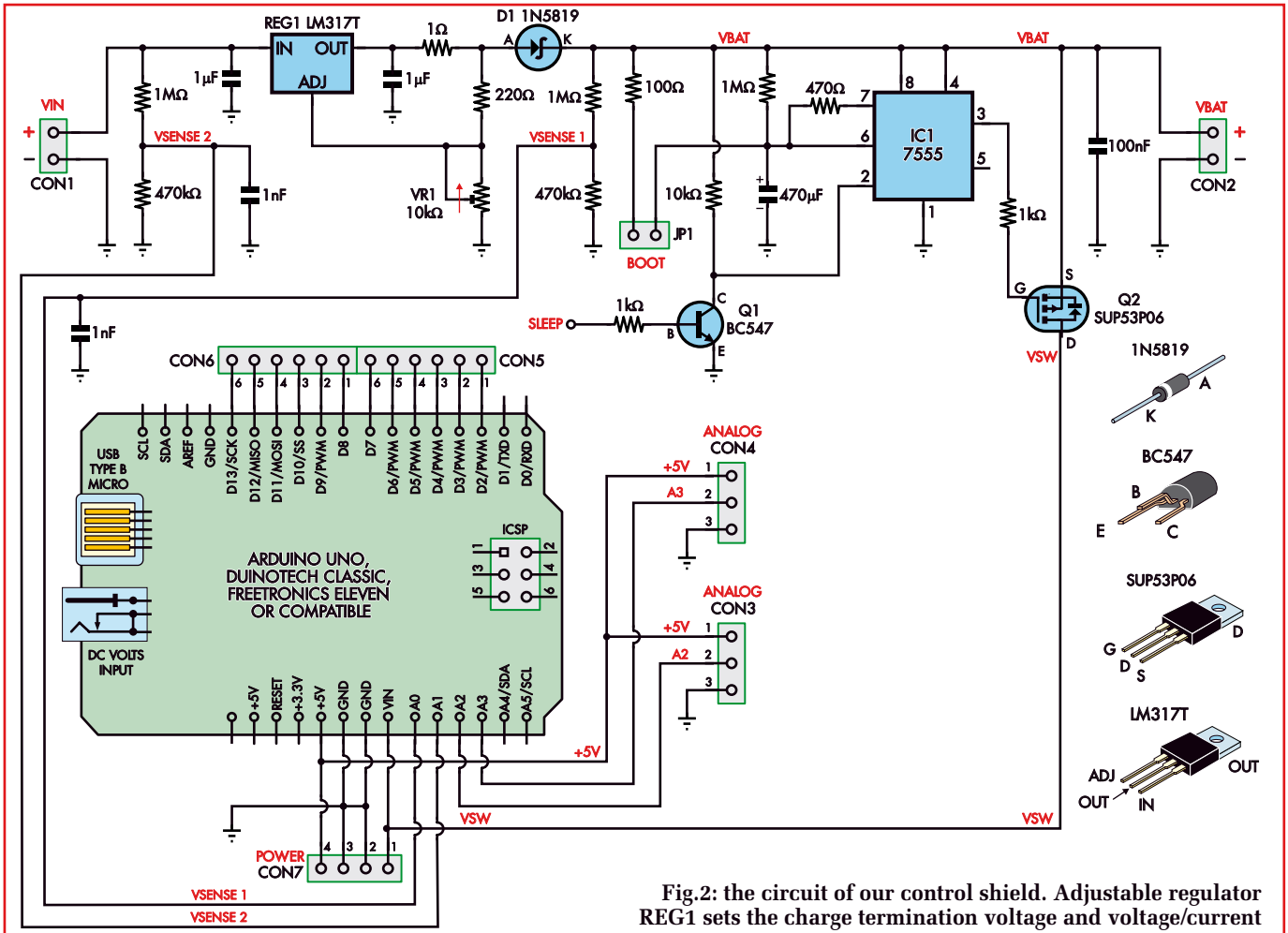


Fig.2: the circuit of our control shield. Adjustable regulator REG1 sets the charge termination voltage and voltage/current characteristics for charging a battery connected via CON2, while IC1, Q1 and Q2 shut down the Arduino for a fixed time before powering it back up again, as a power-saving measure. The effect is that the Arduino is powered intermittently, for as long as necessary to do its monitoring tasks.

SIM7000 POWER CONTROL SHIELD

the timing capacitor to charge.

When the Arduino is shut down, it can no longer keep Q1 switched on, so there is no chance of this state latching.

Thus the cycle continues where it began. The Arduino has no way of turning itself on at a particular time; it just shuts down for the period of the monostable timer.

It's not exactly high precision, but it allows very low power consumption while ensuring that the Arduino is powered up periodically to do whatever it needs to do.

Jumper JP1 allows the monostable circuit to be bypassed. If JP1 is shorted, IC1's threshold pin is pulled above 2/3 of its supply, so Mosfet Q2 is forced on. As long as this jumper remains in place, the Arduino is unable to shut itself down.

This can be used to bypass the sleep mode during testing, or to force the 4G Remote Monitoring Station to operate when deployed.

Sensing external voltages

Two six-way headers, CON5 and CON6, are provided to make connections to the Arduino's digital pins. A small prototyping area with nine pads is also provided.

A pad connecting to SLEEP is placed nearby. This is intended to be connected with one of the digital pins via a short jumper wire, meaning the pin used for shutting down the Arduino is not fixed in hardware, but can be altered. For our prototype, we used D7.

A small four-way header is also broken out for 5V, VIN and GND, since connected sensors or peripherals will need access to power.

Two analog pins are connected to resistive dividers to sense the battery voltage (A0) and incoming supply voltage (A1). The 1MΩ/470kΩ divider means that voltages up to 15.6V can be measured. These high values are chosen to minimise loading (to around 10µA), especially on the battery.

The two 1nF capacitors provide a low source impedance for the analog inputs, as otherwise, these voltage readings would be inaccurate.

Two more analog pins (A2 and A3) are broken out to separate three-way headers (CON3 & CON4), along with ground and 5V. These allow common three-wire analog sensor modules to be connected.

Note that there is nothing about this shield which ties it specifically to the 4G Shield. Any application which re-

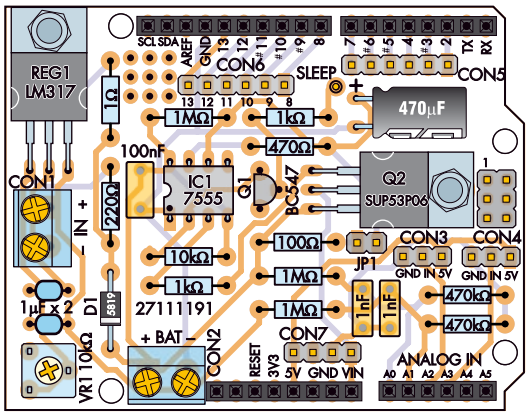
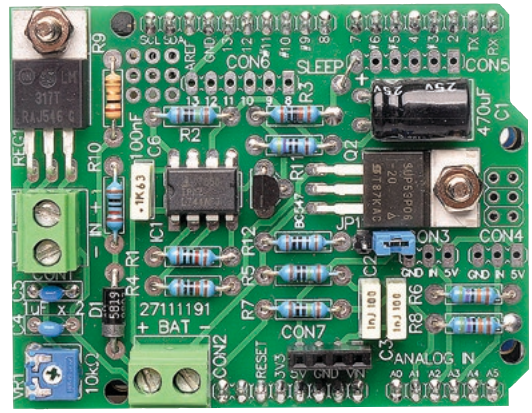


Fig.3: fit the components to the control shield PCB as shown here. It shouldn't take you too long to assemble. Just watch the orientation of diode D1, IC1, Q1 and the electrolytic capacitor. Also, ensure that the wire entry holes for CON1 and CON2 face outwards. You can use standard male headers (fitted to the underside of the board), or stackable headers, depending on how you plan to use the shield.



quires battery charging, monitoring and low power consumption could use this shield.

Building the shield

Use Fig.3, the PCB overlay, as a guide during construction. The shield is built on a double-sided PCB coded 27111191 which measures 53.5 x 68.5mm.

We built our shield with simple headers to plug into the 4G Shield below it.

If you intend to add another shield above this one, you could use stackable headers instead, but that would make it difficult to access the pin headers on top of this board.

Start construction by fitting the resistors. There are several different values, so check the resistance of each with a multimeter. Then solder one of the lead off-cuts between the pad marked "SLEEP" and the Arduino digital pin that you want to use for the shutdown function. We used D7, simply because it is close to the SLEEP terminal and it is not usually used for any specific purpose.

Next, mount the three rectangular MKT capacitors, which are not polarised. The 100nF part may be marked with the code 104 or possibly 0.1µF, while the 1nF parts may be marked 102. Follow with the two 1µF ceramic capacitors, which are also not polarised.

The final capacitor is the low-leakage electrolytic type. It is polarised, and we have left space on the PCB for it to be mounted on its side so that another shield can be fitted above. The negative lead (usually shorter and indicated by a stripe on the can) goes into the pad closer to Q2. If you want to use a larger capacitor for a longer delay, we have left a bit of extra room.

You may wish to apply a small dab of hot glue or neutral-cure silicone sealant to help hold it in place in case the unit is subjected to vibration.

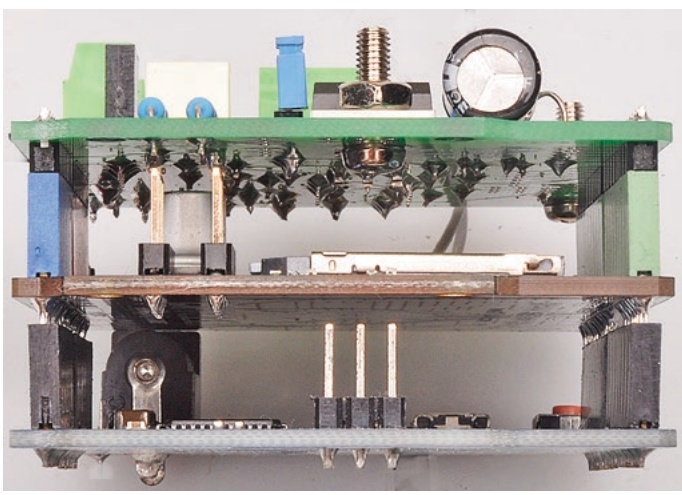
Fit the semiconductors next. D1 is the only diode and goes near CON2, with its cathode stripe closest to CON2. Mount Q1 near the middle of the PCB, orientated a shown. Carefully bend its leads to suit the PCB footprint, push down firmly onto the PCB and solder it in place.

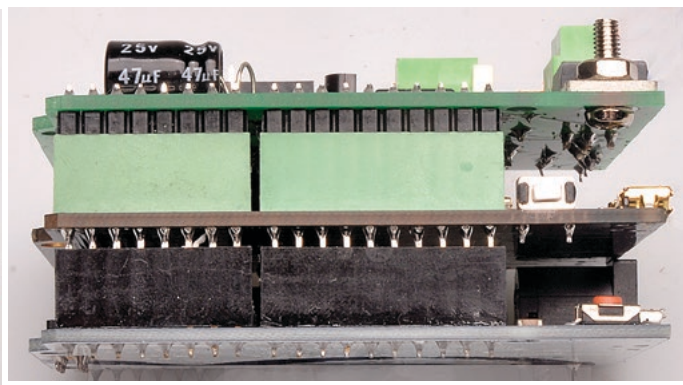
Q2 and REG1 are both in TO-220 packages that are mounted flat against the PCB, to keep the overall height low. Don't get them mixed up. Bend the leads back on each part, around 8mm from where the body meets the leads. Push the leads into the PCB pads and make sure the tab hole lines up with the PCB, then use an M3 machine screw and nut to secure the regulator before soldering its pins.

Next, fit timer IC1. You may need to gently bend the leads inwards to fit the IC to the PCB. Check its orientation to make sure it matches the PCB overlay diagram, then solder two diagonally opposite pins and check that it is flat against the PCB. If not, remelt the solder and adjust, then solder the remaining pins.

To solder the Arduino headers, plug them into another Arduino board (such as the Leonardo) to keep them straight. Place the shield over the pin headers and once you are happy that the headers are straight, solder each pin to the PCB. Plain headers are soldered on top while stackable headers are necessarily soldered underneath. Then remove the shield from the Arduino board.

Now mount CON1 and CON2, the screw terminal connectors. They are identical, and care should be taken





With stackable headers, the three shields (PCBs) simply connect together via their header pins and sockets, as shown here.

that the wire entry holes face out from the PCB.

You can now fit headers for JP1 and CON3-CON7 for breaking out the various Arduino pins. They aren't needed for the most basic usage of the 4G Remote Monitoring Station, but they are handy for adding extra sensors if and when necessary.

We used a four-way female header strip for CON7, to allow the VIN voltage to be monitored easily.

To help test whether the shield is feeding power to the VIN pin, we rigged up a test LED by soldering a 1kΩ resistor to one lead. We then plugged this LED/resistor combo into the VIN/GND pair on CON7, with the LED anode to VIN.

Testing

We can do a few basic tests to check that everything is working as expected. The shield must not be connected to any boards during these tests. Wind VR1 fully anti-clockwise before powering it up.

The first step is to adjust VR1 for the correct battery charging voltage. This is done without a battery connected. To do this, connect a power supply to CON1 which supplies at least 3V more than the fully charged battery voltage. Adjust VR1 until the correct maximum charge voltage is reached at CON2.

For an SLA or similar 12V nominal type battery, set it to around 14.4V. In practice, this voltage is only reached at zero current, so the actual charge voltage is a bit lower than this.

If you are unable to set the voltage correctly, check the components relating to REG1. Otherwise, connect the battery and check that it is charged. You should see the battery voltage rising slowly.

Next, check the voltage between VIN and GND, using the LED we mentioned earlier or a voltmeter. These pins are easily accessible on CON7. You will probably get a zero reading, meaning that Q2 is off.

In this case, if you check the voltage across the electrolytic capacitor, you should find that it is slowly rising. This can be measured at pin 6 of IC1 referred to GND. Note that the load presented by your multimeter might affect this reading.

Now bridge JP1, to force Q2 on, and re-check the VIN voltage. It should be close to the battery voltage and the voltage at pin 6 of IC1 should be low.

To simulate the Arduino activating the switch-off, momentarily connect the SLEEP pad to the 5V pin of CON2 using a 1kΩ resistor. VIN should drop to zero, and the electro should start charging. If it isn't, that could indicate that your capacitor is leaky.

If you're fussy about the exact timing of the sleep period, you can measure the time and change the values of the timing components to tweak it. Keep in mind that the Arduino needs to operate for at least 30 seconds to update its state, so sleep periods shorter than two minutes are not that useful, as the Arduino will spend much too much time starting up.

Once testing is complete, disconnect the power supply and batteries.

Building the Remote Monitoring Station

Having built the shield, now we can put it all together. We chose to use an Arduino Leonardo board for our prototype. It uses the ATmega32U4 micro rather than the Uno's ATmega328. The "U" indicates that this IC supports USB.

Their specs are otherwise quite similar, but the Leonardo has the advantage that the hardware serial port on D0/D1 is not shared with the USB host serial interface used for programming. We can therefore use this to communicate with the SIM7000. The Leonardo also has an extra 512 bytes of RAM; this can be handy for remote monitoring as we need to store and process data before sending it.

If we had used an Arduino Uno, we would have been forced to choose between using the hardware serial port (D0/D1) to communicate with the SIM7000, which would interfere with programming and debugging, or using a software serial port which is slow and has a lot of overhead.

So we set the switch on the SIM7000 shield to the D0/D1 position, and as mentioned above, we used D7 as the sleep control pin.

To set up the 4G Shield, fit the two antennas and a working SIM card. As with many of these sorts of applications, a prepaid SIM is preferred in case the microcontroller 'goes nuts'. With a prepaid limit in place, there is no chance of accidentally racking up huge data or call charges.

Now plug the 4G Shield into the Leonardo and then plug the power control shield on top. Check for any fouling between the shields; if you have not trimmed all the leads closely, they may short together.

Our sample software simply logs data from the onboard sensors. We've also marked some places in the code to add your own tests or actions. For example, you could monitor a voltage and send an SMS if it gets too low or high. Or similarly, you could send an SMS if a switch is opened or closed.

Parts list – 4G Remote Monitoring

1 Arduino Leonardo or compatible board
1 DFRobot SIM7000 shield [Digi-Key/Mouser (Cat DFR0505) or direct from www.dfrobot.com]
1 4G SIM card for SMS and data use
1 power control shield (see below)
1 12V rechargeable battery and suitable charging source (eg, a small 12V solar panel)

Parts for power control shield

1 double-sided PCB coded 27111191, 53.5 x 68.5mm
2 2-way, 5mm-pitch PCB-mount terminal block (CON1, CON2) [Jaycar HM3172, Altronics P2032B]
1 set of Arduino headers (1 x 6-way, 2 x 8-way, 1 x 10-way – see text)
1 2-way male pin header with jumper shunt (JP1)
2 3-way male pin header (CON3,CON4)
2 6-way male pin header (CON5,CON6)
1 4-way female header (CON7)
2 M3 x 6mm machine screws & nuts (for mounting REG1 & Q2)

Semiconductors

1 7555 CMOS timer IC, DIP-8 (IC1)
1 LM317 adjustable voltage regulator, TO-220 (REG1)
1 BC547 NPN transistor, TO-92 (Q1)
1 SUP53P06 P-channel Mosfet, TO-220 (Q2)
1 1N5819 schottky diode (D1)

Capacitors

1 470µF 25V low-leakage electrolytic
2 1µF multi-layer ceramic [Jaycar RC5499]
1 100nF MKT
2 1nF MKT

Resistors (all ¼ W 1% metal film)

3 1MΩ	2 470kΩ	1 10kΩ
2 1kΩ	1 470Ω	1 220Ω
1 100Ω	1 1Ω	

1 10kΩ mini horizontal trimpot (VR1)

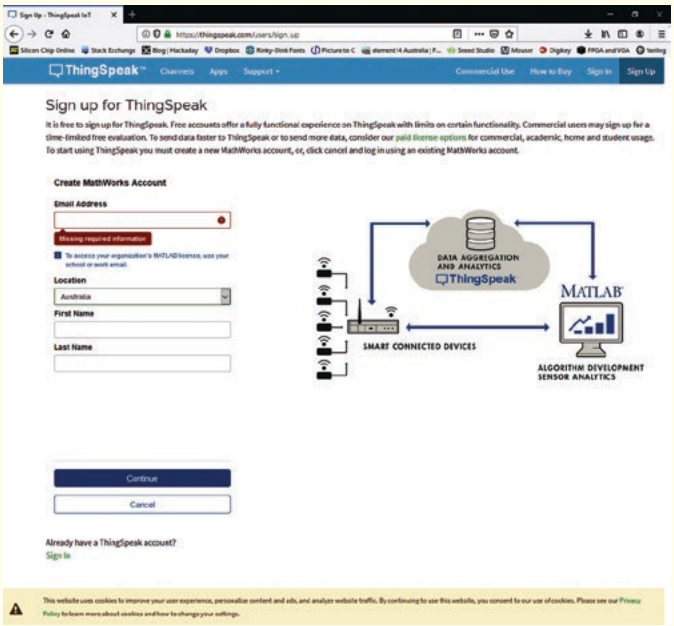


Fig.4: apply for a ThingSpeak account via the web page shown here. This is needed to use the software we've written, as ThingSpeak lets you upload data to "the cloud". MATLAB users can use their existing account for ThingSpeak.

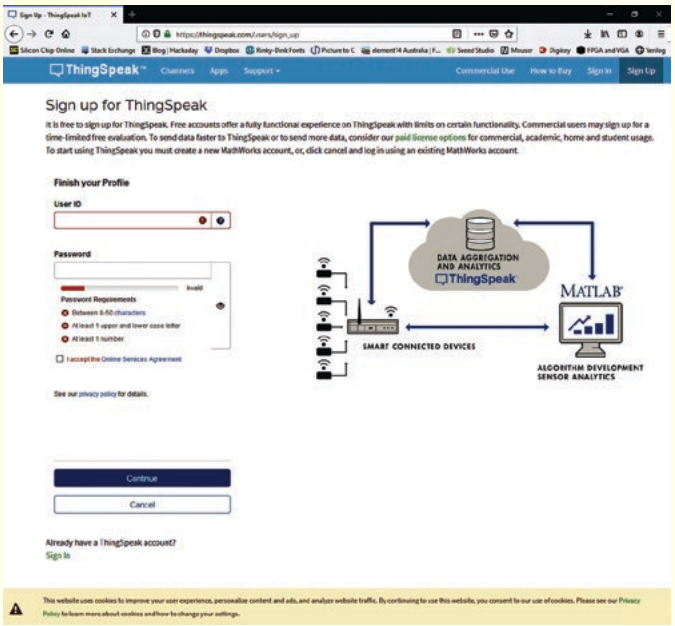


Fig.5: as with many online services, you need to create a username and password for ThingSpeak. This page indicates if your chosen username is free, and how strong it thinks your password is.

You will need to set up a ThingSpeak account to make full use of our sample code.

Setting up a ThingSpeak account

ThingSpeak can be accessed for commercial use with a time-limited free period, but a free license is available for personal use and offers four 'channels' and up to three million updates per year. If we were to send an update every ten minutes, then we would only need around 50,000 updates per year.

Go to https://thingspeak.com/users/sign_up and enter the information as shown on Fig.4. You may be prompted to confirm that you wish to use a personal email address, and also to click on a link sent in an email to verify that email address.

Once this is done, create a user ID and password, accept the Online Services Agreement and click continue as per Fig.5. You will be prompted to select how you will use ThingSpeak. To be able to use the free license, you should choose "Personal, non-commercial projects".

The next step is to create a channel. Each channel consists of up to eight fields, so in theory, you could have up to four 4G Remote Monitoring Stations, each writing to their own independent channel.

Click on "New Channel" and fill out the information as shown in Fig.6. You don't need to use all eight fields, but we have set the Arduino software to use all eight as shown. You should use the same fields unless you plan to modify the software.

Click Save, and you are shown the Channel data on the Channel management page, as seen in Fig.7.

Note that we did not create fields for latitude and longitude. ThingSpeak has hidden fields for this information. It can't be seen on the graphs, but is downloaded in the CSV data. Our Arduino code logs latitude and longitude to these hidden fields.

API keys

To allow our device (and only our device) to upload data to our channels, we need an API key. It must be programmed into the Arduino code for your 4G Remote Monitoring Station to work with your ThingSpeak channel. Copy the 16-character alphanumeric code under "Write API Key" to somewhere safe; we'll add this to the Arduino code soon.

You can test that your channel is working by copying the text after the word "GET" in the "Write a Channel Feed" box. Paste this into a web browser and hit Enter; you should see a blank page with the number "1".

This indicates that this is the first update, and shows how the 4G Remote Monitoring Station uploads data to ThingSpeak. This only updates one field; if you are familiar with HTTP, you might want to experiment with this.

Browse back to the "Private View" of the created channel, and you should see some activity in the first field; this is the data you sent from the web browser. You can leave this window open while testing, as it will update in near-realtime and you can see the results.

Arduino libraries

There are four libraries needed for the software we have written; two are included with most Arduino IDE distributions. We used version 1.8.5 of the Arduino IDE.

The avr/sleep and Wire libraries are the two usually included. The first library provides functions for low-power modes, while the second provides an I^2C interface for communicating with the BME280 sensor.

The third library, which we created, is named "cwrite". It lets us read and write from a character array as though it is a stream object, so we can use the print function's ability to format floating-point numbers to form a URL.

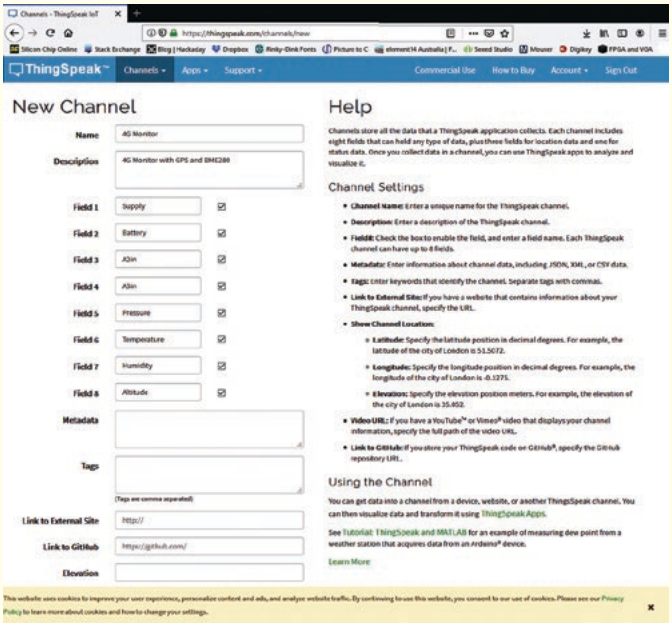


Fig.6: we recommend that you (at least initially) create a ThingSpeak channel and set up its fields as shown here. These fields suit the data produced by the 4G Remote Monitoring Station software. They can be changed later if necessary.

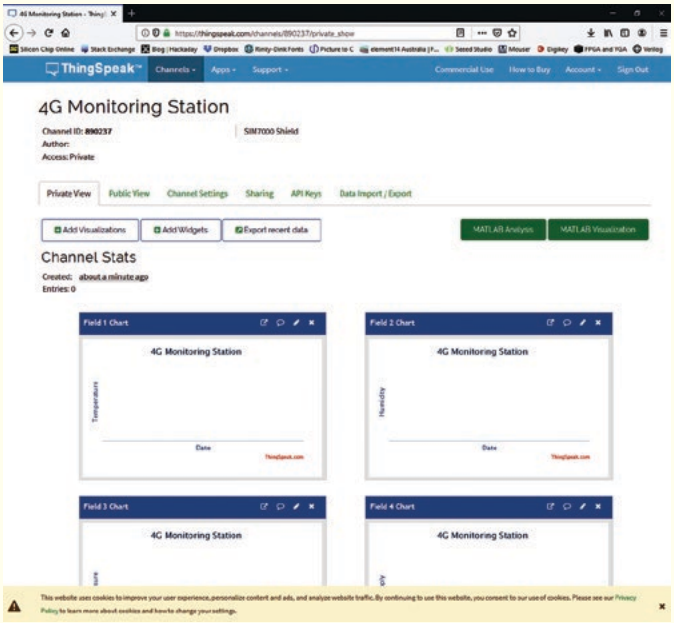


Fig.7: once the channel has been created, you can go to its overview, which defaults to a series of charts. You can add more with the "Add" buttons. By default, the channel data is private, but you can set it to be visible to others if you'd like to.

The resulting datum can then be sent to the 4G Shield in one go.

This library can be seen as two extra tabs in the Arduino project. If you make a copy of the project (by using File -> Save As...), then this library is copied too.

The final library is to make the actual temperature, humidity and pressure readings from the BME280 sensor.

It is written by a company called SparkFun and can be installed via the Library Manager utility of the Arduino IDE. Search for "sparkfun BME280" under the Library Manager and click Install.

We have included this library in our software bundle for this project, in case you can't find it.

Arduino software

We set up the Arduino software to work with the eight fields that we have just created, plus three hidden fields of latitude, longitude and height above sea level.

These three fields are from by the GNSS receiver on the SIM7000 module, plus the BME280's atmospheric pressure sensor to determine altitude.

The software is modularised in such a way that proficient Arduino users can modify it to behave differently, if needed.

In any case, you will need to edit the software to suit your API key. Around line 28, change the text API_KEY_HERE to the API key that you copied earlier. You should end up with a 16-character sequence surrounded by double quotes.

Below this, on lines 29 and 30, are entries for phone numbers. Any incoming text messages have their number checked against the AUTH_NUMBER string. The sequence AUTH_NUMBER_HERE should be replaced by the trailing digits of your phone number.

We have done it this way to allow matching of both national and internationally formatted numbers. Thus for an Australian mobile number, the first digit should be the '4', meaning the leading '0' is dropped.

The sketch simply matches whatever digits are present. So if this were changed to "693", then any number ending in "693" would be accepted. If you don't wish to use this feature, leave it as the default string, as this is highly unlikely to match an incoming number.

The outbound number should be a fully-qualified international mobile number; eg, an Australian mobile phone number including country code would start with "+614" followed by eight digits. This is used for outgoing text message alerts.

Many of the other 'defines' are set to set things like the analog input voltage measurement ratios, and how much memory is available. There is no reason to change these unless you are modifying the hardware.

The software performs basic initialisation of the hardware in the setup() routine. More initialisation happens in the loop() function, particularly for the 4G Shield.

The code sends the shield some data to see if it is powered up and if not, toggles the power line.

A set of fixed sequences are sent to put the 4G Shield in a known state. The shield is given ten seconds to get a GNSS fix. If this is successful, the unit's speed is checked, and a message is sent if it is higher than 100km/h. This is a basic demonstration of how easily an action can be performed based on sensor state.

The code then sends an update to ThingSpeak; this is wrapped up in a single function which validates the GNSS data and only sends that data if it is valid.

The Arduino then checks for text messages from the authorised number. If one is found, a canned response is sent.

You can modify the code to check the message content and perform different actions (and supply different

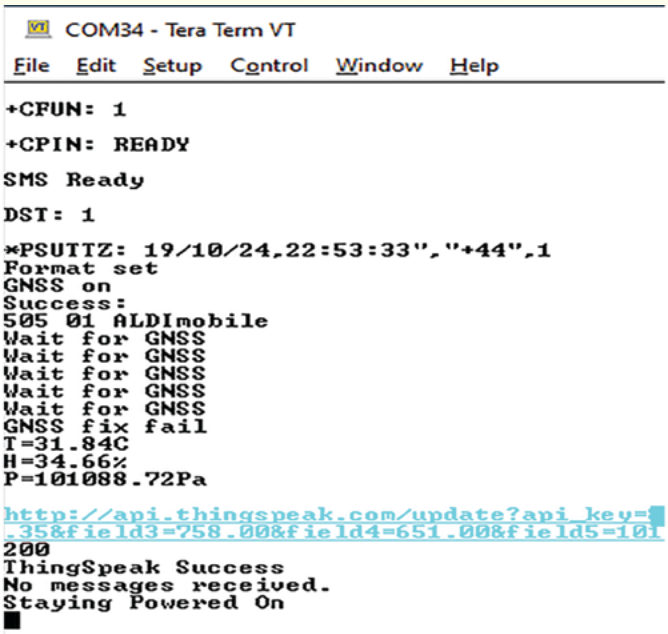


Fig.8: here is some sample debugging data from the serial port of the Remote Monitoring Station during normal operation. Your data may differ, if, for example, you have a different telco.

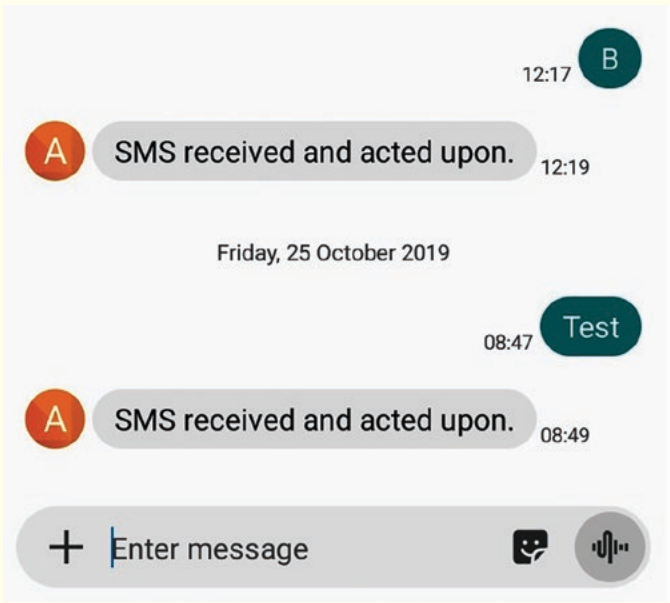


Fig.9: as the 4G Remote Monitoring Station is only powered up infrequently, it may not respond to an SMS command immediately. The two-minute delay shown here occurred during testing, when we had the Interface Shield set to power down for about one minute at a time. The canned response can be changed by editing the Arduino code.

responses) depending on it.

If the GNSS data is not valid, then instead of powering off, the Arduino goes to sleep and leaves the 4G Shield running to allow it to get a fix. This does not reduce power as much as switching the Arduino off, but does give it a chance to get a position fix.

If the GNSS data is valid, then the modem's power pin is toggled (to perform a controlled shutdown), and D7 is driven high to power down everything else.

Next time the Arduino powers back up, the sequence repeats, resulting in ThingSpeak updates about every 10 minutes.

Each update uses around 2kB of data, which, according to our mobile plan, costs around $0.0001. During our testing, we sent around 6000 updates (over a month worth of updates) for a total cost of $1.08. Your plan might vary.

Finishing it off

Connect the Arduino board to your computer and select the Leonardo board and its corresponding serial port from the Arduino IDE menus.

Compile and upload the "4G_Monitoring_Station.ino" sketch. Unplug the Leonardo and attach the two shields.

Connect the battery to CON2 and the power source to CON1. Briefly short out JP1 and check that the whole assembly powers up.

The upload to ThingSpeak should take less than a minute. If it does not, then you may need to do some debugging to find out what's wrong.

Re-connect the Leonardo board to the computer and open a serial terminal program to monitor the output. It should look like that shown in Fig.8. Look for a 200 HTTP code and "ThingSpeak Success" message.

If you get this, then uploads to ThingSpeak are working correctly.

You might find that the Arduino Serial Monitor does not behave well when the Leonardo powers down. We had success with a program called TeraTerm, as this automatically reconnects to the serial port if it disconnects and reconnects.

Unfortunately, the USB lead will also power the Leonardo, so the power down functions may not work as expected while connected to a computer.

Here's the USB Port Protector from our May 2018 issue – we used one of these without components, except for the USB plug and socket, during testing. This allows data to be transferred, but not power.

A trick for testing

To test our prototype, we needed a way to allow USB communication but without powering the Leonardo, as this interferes with the power control hardware.

To achieve this, we used one of our USB Port Protector PCBs described in May 2018 (siliconchip.com.au/Article/11065).

If the Port Protector PCB is wired up with no components except the USB plug and USB socket (CON1 and CON2), then it connects GND, D+ and D-, but not 5V.

Thus this 'dongle' can be used to connect a USB device to allow data but not power to be transferred. The Leonardo is then powered via its onboard 5V regulator fed from the VIN pin.

Take care that the ground of your computer is not at a different potential to the ground of the 4G Remote Monitoring Station; for example, if you are powering it from a bench supply or similar, make sure the outputs are floating.

You can use a battery-powered com-

puter for testing if you are not sure about this.

Debugging

While the code is quite complex, we did not run into many problems with it. But in case you do, we'll run through some of the error messages the Arduino might display.

If you don't see the "GNSS on" or "Format set" messages, your Arduino is probably not communicating with the 4G Shield.

According to the shield's data sheet, it communicates at 115,200 baud, but our unit was set to 19,200 baud. You can change this setting at line 5 in the Arduino code.

After the "GNSS on" message, you should see "Success" and a network name.

If you see "Fail" here, the 4G Shield is not registering with the network. This generally happens when the 4G Shield has no reception. It could be due to the shield not supporting your telco's frequency band, or you may be out of range of a cell tower. Check that the antennas are connected correctly.

You will occasionally see "GNSS fix fail" as the 4G Remote Monitoring Station compromises getting a fix at all times for saving power.

The code tries to retrieve the APN (access point name) from the 4G Shield and use it to connect to mobile data. If you see a message referring to APN, CSTT or bearer failing, then this is not being set up correctly. Check the APN name that your provider uses.

The URL that the 4G Remote Monitoring Station uses should be displayed, followed by an HTTP result code.

If it is not 200 (HTTP success), check https://au.mathworks.com/help/thingspeak/error-codes.html to see what other error codes mean.

If it still isn't working, numerous extra debugging lines in the code have been commented out (by adding "//" to the start). You can enable the extra messages by removing these and compiling and uploading the code again.

You can also try our "Leo_FTDI_with_passthrough.ino" sketch. This configures the Leonardo to allow direct communications between the serial port and the 4G Shield.

You can try different baud rates to see what works and send commands directly to the 4G Shield.

Upload this to the Leonardo and short JP1 on the power control shield. You may need to press the 4G Shield's BOOT button to power it up manually. Once you have confirmed the correct baud rate, upload "4G_Monitoring_Station.ino"' to the Leonardo again.

Conclusion

We've deliberately left this as an open-ended project; we expect that readers will customise the hardware and code to suit different applications.

For outdoor use, we recommend housing everything in an IP-rated plastic enclosure, with both antennas mounted on the underside of the lid.

Including some vent holes, facing down, can help to drain any condensation which may form, and allow the outside air to be sampled by the BME280 sensor. SC