# G2 Mk2 Front Panel Implementation Notes

This documentation is for the G2 mk2 front panel PCB using the Arduino Nano Every processor and 2 MCP23S17 ICs. It has I2C interface to the Raspberry Pi.

# Control Layout

A white rectangle with black lines

Description automatically generated

## Encoder Numbers

The VFO encoder is treated differently from the dual encoders because its inputs are processed differently. From a software perspective, each dual encoder can have A (upper) and B (lower) encoders plus a “click” function. The “click” is treated as a pushbutton by the Arduino.

The top right encoder will be able to have two functions, using a “Shift” button. The Shift functionality is implemented in the Arduino, and the Raspberry Pi will see 6 dual connected encoders.

A white rectangle with black lines

Description automatically generated

Figure 1: Encoder numbers

Normal dual shaft encoders have encoder numbers as follows:

|  |  |  |
| --- | --- | --- |
| **Encoder** | **Reported Number** | |
| **Upper knob** | **Lower knob** |
| 1 | 0 | 1 |
| 2 | 2 | 3 |
| 3 | 4 | 5 |
| 4 | 6 | 7 |
| 5 | 8 | 9 |
| 6 (encoder 5 when shift is active) | 10 | 11 |

## Pushbutton Numbers

Pushbuttons have two numbers: the software scan code from the matrix algorithm; and the number that is reported to the Raspberry pi. A lookup table from scan code gives the reporting number. The scan codes are only used by the Arduino software and have no user meaning.

Band buttons in the 3x4 matrix have two functions according to the BAND shift. The band shift functionality is implemented in the Arduino; the effect is that the user software “sees” additional pushbuttons for band and unshifted function. If shift pressed, a button code 13 higher is given for example "Fil -" will give a code of 18 (no shift) or 31 (band function shifted). The shift buttons themselves do not send codes to the Raspberry Pi. PB13 (encoder 5 pushbutton) will generate a code of 26 if the encoder shift is selected. If encoder shift is active, the button for encoder 5 will generate a different code of 41.

A white rectangular object with black lines

Description automatically generated

Figure 2: Software Scan Codes

A complete list of reported button codes is below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Button Function** | **Code** | **Button Function** | **Code** | **Button Function** | **Code** | **Button Function** | **Code** |
| No button | 0 | RIT/XIT | 11 | Split | 22 | 17M | 33 |
| Encoder 1 press | 1 | Encoder 4 press | 12 | User 1 | 23 | 15M | 34 |
| Encoder 2 press | 2 | Encoder 5 press | 13 | User 2 | 24 | 12M | 35 |
| Encoder 3 press | 3 | Mode + | 14 | User 3 | 25 | 10M | 36 |
| ATU | 4 | Filter + | 15 | No button | 26 | 6M | 37 |
| 2 Tone | 5 | Band + | 16 | 160M | 27 | LF/MF | 38 |
| Tune | 6 | Mode - | 17 | 80M | 28 | Band shift\* | 39 |
| MOX | 7 | Filter - | 18 | 60M | 29 | Encoder shift\* | 40 |
| CTUNE | 8 | Band - | 19 | 40M | 30 | Shifted Encoder 6 press | 41 |
| LOCK | 9 | A>B | 20 | 30M | 31 |  |  |
| A/B | 10 | B>A | 21 | 20M | 32 |  |  |

Band and encoder shift codes are only generated if a command has been sent to disable normal shift processing. This is appropriate for Thetis but not for piHPSDR.

## Indicator Numbers

Note LED10, 11 are not accessible to client s/w. Their functions are controlled by the Arduino, unless shift functionality has been disabled by a command.

A close-up of a white rectangle

Description automatically generated

Figure 3: Indicator Numbers

# Device, Pin Allocations

The I/O entities to be interfaced are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface** | **Devices** | **Inputs** | **Outputs** |
| VFO encoder | 1 | 2 |  |
| Dual encoder | 5 | 20 |  |
| Switch/pushbutton | 22 + 5 | 8 | 5 |
| LED | 11 |  | 11 |
| SPI Chip selects |  |  | 2 |
| Interrupt output |  |  | 1 |

## MCP23S17 Pins

|  |  |  |  |
| --- | --- | --- | --- |
| **MCP23S17** | **Number:** | **1** | **A2=0 A1=0 A0=0** |
| **Port A** | **Encoder inputs** | **Port B** | **Encoder inputs** |
| GPA7 | Encoder 1 upper A | GPB7 | Encoder 3 upper A |
| GPA6 | Encoder 1 upper B | GPB6 | Encoder 3 upper B |
| GPA5 | Encoder 1 lower A | GPB5 | Encoder 3 lower A |
| GPA4. | Encoder 1 lower B | GPB4 | Encoder 3 lower B |
| GPA3 | Encoder 2 upper A | GPB3 | Encoder 4 upper A |
| GPA2 | Encoder 2 upper B | GPB2 | Encoder 4 upper B |
| GPA1 | Encoder 2 lower A | GPB1 | Encoder 4 lower A |
| GPA0 | Encoder 2 lower B | GPB0 | Encoder 4 lower B |

|  |  |  |  |
| --- | --- | --- | --- |
| **MCP23S17** | **Number:** | **2** | **A2=0 A1=0 A0=1** |
| **Port A** | **Switch matrix column/ enc** | **Port B** | **Switch Matrix Row INPUT** |
| GPA7 | LED1 | GPB7 | Switch Matrix Row 8 |
| GPA6 | LED2 | GPB6 | Switch Matrix Row 7 |
| GPA5 | LED3 | GPB5 | Switch Matrix Row 6 |
| GPA4 | LED4 | GPB4 | Switch Matrix Row 5 |
| GPA3 | Switch Matrix Column 4 out | GPB3 | Switch Matrix Row 4 |
| GPA2 | Switch Matrix Column 3 out | GPB2 | Switch Matrix Row 3 |
| GPA1 | Switch Matrix Column 2 out | GPB1 | Switch Matrix Row 2 |
| GPA0 | Switch Matrix Column 1 out | GPB0 | Switch Matrix Row 1 |

## Arduino Pins

|  |  |  |  |
| --- | --- | --- | --- |
| **Arduino:** | **Arduino Nano Every** |  |  |
| DIG0 / TX | VFO encoder A | DIG11 | SPI Controller Out, Peripheral In |
| DIG1 / RX | VFO encoder B | DIG12 | SPI Controller In, Peripheral out |
| DIG2 | Encoder 5 upper A | DIG13 | SPI Clock |
| DIG3 | Encoder 5 upper B | A0 | LED10 |
| DIG4 | Encoder 5 lower A | A1 | LED11 |
| DIG5 | Encoder 5 lower B | A2 | MCP23S17A (U1) CS~ |
| DIG6 | LED5 | A3 | MCP23S17B (U2) CS~ |
| DIG7 | LED6 | A4 / SDA | I2C SDA |
| DIG8 | LED7 | A5 / SCL | I2C SCL |
| DIG9 | LED8 | A6 |  |
| DIG10 | LED9 | A7 | Interrupt to Raspberry pi (GPIO15) |

# Arduino Software Structure

An 8 bit Arduino Nano Every will be used, with software substantially based on the Andromeda panel code.

## Concept for Operation

The Arduino will interface the I/O devices, consisting of optical VFO encoder, mechanical dual shaft encoders, pushbuttons and LEDs. It will handle all debouncing and present completed event data to the Raspberry pi. Events will be stored in a simple circular buffer queue and transferred at a timestep of every few milliseconds (potentially every 10ms) to avoid signalling and data transfer becoming a drain on processor resources.

Events will consist of:

* VFO encoder steps (potentially up to 20 steps per 10ms timestep);
* Mechanical encoder steps;
* Pushbutton events.

In addition the Arduino will receive commands at similar rate via I2C to set front panel LEDs.

A diagram of a computer program

Description automatically generated

Figure 4: Software Structure

The VFO encoder may generate 500 steps per revolution and can be turned at 4 revolutions per second; so several steps per timestep are possible. VFO encoders generate clean edges, and interrupt driven operation is viable.

Mechanical encoders generate noisy edges and will be polled, with a debounce period of around 5 - 10ms after detection of a transition. A high turn rate would be one revolution per second and typically 24 steps so typically a rate of 40ms per step. One step per event is sufficient. All encoders will be scanned in parallel, but it is unlikely that more than one will be turned at a time so a high event rate is unlikely.

Pushbutton encoders also generate noisy edges and will be polled. The pushbuttons will be scanned in a matrix, allowing only one button press at a time. A debounce period of ~10ms is required and the event rate will be a max of say two presses per second. Both “normal” and “long” presses will be decoded.

Two “special” pushbuttons will provide a shift function. These will provide additional functions for 12 pushbuttons, and for one dual encoder. This will be decoded by the Arduino and the Raspberry pi will see an interface as if an additional 12 buttons and one additional dual encoder are present. (These two buttons and their LEDs can also be set to operate like other pushbuttons by command message from the Raspberry pi).

## Use of Timeslots

Like all of my Arduino projects, the code will work to a fixed timer driven timestep. It will not follow the normal Arduino practice of executing a loop continuously; it will only run a loop once per timestep. That allows real time operation with known event timings.

Follow the principle from the Andromeda front panel controller: 2ms timestep is the starting point.

@400KHz, byte read over I2C ~45us. SPI could be 10x faster.

In one timeslot we need to:

* Do one update of the key matrix (involves SPI read, then SPI write);
* Update half of the encoders; (I2C 16 bit read);
* Update any LEDs;
* Add any events to event queue and assert interrupt if it is not empty;
* Drive new switch matrix column in advance of next tick.

The I2C operations will be interrupt driven and will occur at any time in the sequence; accessing the circular buffer event queue will need to be wrapped with disabled interrupts.

## Message Transfer to/from Pi

### I2C Transfers

The Arduino will interrupt the Raspberry pi via a GPIO pin if there is any data in the event queue; the interrupt will be removed when the queue has been emptied (therefore it is level triggered). The raspberry pi will be able to read event data as 16 bit words via I2C; it will read words until the queue is empty. the Arduino will transfer a zero word if there is nothing in the queue. The Raspberry pi will be able to write new LED data at any time. A message queue of 15 words is likely to be sufficient, allowing over 100ms latency in the Raspberry pi.

The pi could also poll the Arduino by reading the I2C port every 10ms or so and taking all messages when it finds data present.

A slave write I2C transfer is a 3 byte sequence: 8 bit address / RW word, 8 bit data low byte, 8 bit data high byte.

A slave read I2C consists of a 1 byte address written to the Arduino, then a 2 byte read (low byte 1st)

Be aware of the clock stretching issues which is an apparent bug in the Raspberry pi (see 6.3). This means the 1st bit transferred back from Arduino to pi on a read can be lost: ie bit 7 of the low byte. A pragmatic “fix” is to avoid using that bit.

### Message Structure From Arduino

|  |  |  |
| --- | --- | --- |
| **Address** | **Register bits 15:8** | **Register bits 7:0** |
| 0x0A | LED Word 15:8 | LED Word 7:0 |
| 0x0B | 3:0: Event ID  7:4: queue depth after this read | 7:0 Event data |
| 0x0C | Product ID. 0x03 (G2V2 front panel) | SW version |
| Other | 0x00 | 0x00 |

|  |  |  |
| --- | --- | --- |
| **Field** | **Bits** | **Meaning** |
| Queue Depth | 4 | Number of events held in the queue (0-15) before this transfer. 0 means no data.  A value of 1 will leave no remaining data in the queue after this read. |
| Event ID | 4 | Event type  0: no event  1: VFO encoder step  2: dual encoder step  3: pushbutton press  4: pushbutton long press  5: pushbutton release  6-15: reserved |
| Event Data | 8 | Pushbuttons: scan code for pushbutton (1 to 127)  VFO encoder: no. steps (signed 7 bits: -64 to +63)  Dual encoder: bits (6:3) = encoder number; 2:0 = step count (-4 to +3) |

### Message Structure to Arduino

|  |  |  |
| --- | --- | --- |
| **Address** | **Register bits 7:0** | **Register bits 15:8** |
| 0x0A | LED Word 7:0 | LED Word 15:8 |
| Other | ignored | |

A simple command word to the Arduino is sufficient.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Meaning  (normal) | PBX | 0 | 0 | 0 | 0 | LED11 | LED10 | LED9 | LED8 | LED7 | LED6 | LED5 | LED4 | LED3 | LED2 | LED1 |

|  |  |
| --- | --- |
| **Field** | **Meaning** |
| LED(10:0) | Control the state of individual LEDs. 1 if lit. bit0 sets front panel LED1  Note LEDs 10 & 11 are not controlled by this message unless PBX=1 |
| LED(15): PBX | Shift Button Override. If set, the two shift buttons and their LEDs operate like the others i.e. do not control the scan codes generated by other buttons and encoders. |

## I2C interface

There is an I2C interface to the Raspberry Pi. The Pi is the bus master, and the Arduino responds as a slave at address 0x15. The pi will be able to establish whether the Arduino is present to determine if there is an attached front panel (and to distinguish from the current “controller 2” panel).

I2C uses Arduino pins A4 & A5; pullups are required.

An interrupt pin will be required to signal to the Pi that data is available. This is active low, and open collector (requiring pullup to 3.3V by the pi). Open collector simulated by setting pin to be an output only if int is to be asserted.

I2C needs to support the following operations:

* 16 bit slave read, initiated by the Raspberry pi
  + Arduino returns event 16 bit queue entry, or 0 if no event available
  + Arduino clears the interrupt if the event queue is now empty
* 8 bit slave write, initiated by the Raspberry pi
  + Arduino sets LEDs and one control bit

Two registers will be provided for writing LEDs.

Four registers will be provided for read: to read back the LED registers; read board type and s/w version; and read button/encoder events.

Note that the I2C interface on the prototype connects to the Raspberry pi which has 3.3V logic levels. There are two ways forward:

1. Use level translators between levels;
2. Use direct wired connection, with the (internal to pi) pullups to 3.3V
   1. This means the Arduino gets max input of 3.3V, and VIH is 3.5 so not guaranteed to work).

## SPI ports

SPI connects to two MCP23S17 devices. Each has an active low chip select input, to be driven by the Arduino. There is no interaction or dependency with the Raspberry Pi interface.

SPI needs to support the following operations:

1. 8 bit register write, for configuration & for LED / matrix column write
2. 16 bit read from adjacent registers (encoder data read)
3. 8 bit read from one register (switch matrix row read)

# Software Implementation

## Arduino Issues

Arduino Nano Every has an 8 bit ATMEGA4809 processor. I need to change some libraries

* EEPROM.h
* Timer – new code will be needed; no known libraries for ‘4809
* You have to install “Arduino Mega AVR Boards” in the Arduino console Boards manager
* Interrupt driven VFO encoder library doesn’t support ‘4809

Interrupt driven code is poor at debouncing. It does work well with bounce-free optical encoders.

Modified ClickEncoder works well for the other “mechanical” encoders. I’m already using modified encoder code, not in a library. Should be simple to change it again to pass 2 bits of data into it, rather than having I/O pin numbers passed to it.

|  |  |
| --- | --- |
| D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 GND RST D0 D1  A close-up of a blue circuit board  Description automatically generated  D13 3V3 REF A0 A1 A2 A3 A4 A5 A6 A7 5V RST GND VIN | A blue circuit board with white text  Description automatically generated |

MCP23S17 are 3mA I/O: so not suitable for driving LEDs in a matrix.

The processor is only 8 bit: so change variables wherever possible to be byte (8 bit unsigned) int8\_t (signed 8 bit) or int (16 bit) rather than long (32 bit)

## Optical Encoder Scanning

The Broadcom type encoder gives 120 pulses per revolution. By counting every edge, ie 480 edges per revolution, it is good enough for VFO control (4.8KHz per revolution @10Hz step).

The ball bearing encoders are too fast. The 600ppr encoder gives 2400 edges per revolution and when turned at about 1.5 turns per second, the VFO control on Thetis starts to run backwards.

A solution is to have conditional compilation, and for the high resolution optical encoders only interrupt on the rising edge of one input (therefore 600 interrupts per revolution) and use the other input to sense direction.

In globalinclude.h:

To compile for the high resolution encoder:

#define HIRESOPTICALENCODER 1

To compile for the Broadcom type encoder:

//#define HIRESOPTICALENCODER 1

## LEDs

Simple lookup from the software number to a CPU pin

|  |  |  |
| --- | --- | --- |
| **Arduino pin** | **s/w number** | **Function** |
| Dig 2 | 1 | MOX |
| Dig 3 | 2 | Tune |
| Dig 4 | 3 | 2 Tone |
| Dig 5 | 4 | ATU Tune solution |
| Dig 6 | 5 | ATU enabled |
| Dig 7 | 6 | XIT |
| Dig 8 | 7 | RIT |
| Dig 9 | 8 | A/B |
| Dig 10 | 9 | VFO lock |
| A0 | 10 | Band shift |
| A1 | 11 | Encoder shift |

## Keypad scanning

Pushbuttons are scanned using a 4x8 matrix allowing a possible 32 pushbuttons.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Column 4**  **2GPA3** | **Column 3**  **2GPA2** | **Column 2**  **2GPA1** | **Column 1**  **2GPA0** |
| **Row 8**  **2GPB7** |  | Band + |  |  |
| **Row 7**  **2GPB6** |  | Filter + |  | Encoder 3 U/L |
| **Row 6**  **2GPB5** | Encoder 5 U/L | Band - | Mode + | Encoder 2 U/L |
| **Row 5**  **2GPB4** | Encoder 4 U/L | Filter - | Mode - | Encoder 1 U/L |
| **Row 4**  **2GPB3** | Encoder Shift | SPLIT | A>B | MOX |
| **Row 3**  **2GPB2** | RIT/XIT | B>A | User1 | Tune |
| **Row 2**  **2GPB1** | A/B | User3 | Band SHIFT | 2 Tone |
| **Row 1**  **2GPB0** | VFO Lock | USER2 | CTUNE | ATU |

Switches have been placed into the matrix so the columns are approximately geographical with column 1 covering the left of the panel and column 4 the right.

Switch scan code is (row number-1)+ (Column number-1) \*8

I will need to use a simple sequencer to scan the pushbuttons. Don’t attempt to cope with more than one button press. Assert a new column low every software tick, then read the rows and look for a row with one or more bits at zero. Only one row driven low at a time. A “helper” function reads the row input and assigns a row code: 0: no button pressed; 1-8: row 0-7 pressed; FF: more than one pressed.

Columns driven using pseudo open drain outputs from MCP23S17, so if there are shorted columns it doesn’t matter. Outputs only drive a logic 0 level; inactive column outputs are disabled by turning the pins into inputs.

If the same key was pressed for more than 2 seconds, a “long press” is declared in the BUTTON PRESSED state.



Figure 5: Suggested keypad scanning sequencer

## Button Handling

if (shift button override)

{

button code looked up from 1st half of table

Send button code

}

else

{

if (Button == band shift)

{

Toggle BandShift;

Update band shift LED;

}

else if (button == encoder shift)

{

Toggle EncoderShift;

Update encoder shift LED;

}

else

{

if(BandShift)

Lookup PB number from 2nd table

else

Lookup PB number from 1st table

Send button code

}

}

## Switch Matrix Wiring

(note only 4 columns in this case)



Every row has a pullup resistor. Columns are driven by the MCP23017; one column will be 0, the others will be 1. If no buttons pressed, the Row word reads out all 1s ie 0xFF. An “open drain” output is synthesised by enabling or disabling the output; and the output is always driven to zero. Only one column output will be enabled as an output at a time.

## Event Queue

The event queue will be a circular buffer, implemented as an array of 16 bit unsigned ints. A write pointer will indicate where the next write will occur; a read pointer will indicate where the next read will occur. A buffer length of 16 will be sufficient. The current number of entries can be obtained from (W-R) % 16

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row |  | | | | | | | |
| 15 | Initially empty |  | After 3 Writes |  | After 3 reads |  | Later: 3 unread entries; write pointer has wrapped around |  |
| 14 |  |  |  | R |
| 13 |  |  |  |  |
| 12 |  |  |  |  |
| 11 |  |  |  |  |
| 10 |  |  |  |  |
| 9 |  |  |  |  |
| 8 |  |  |  |  |
| 7 |  |  |  |  |
| 6 |  |  |  |  |
| 5 |  |  |  |  |
| 4 |  |  |  |  |
| 3 |  | W | W, R |  |
| 2 |  |  |  |  |
| 1 |  |  |  | W |
| 0 | W, R | R |  |  |

# CAT Commands

This unit uses the same CAT commands as used for Andromeda. That means the existing THETIS command set can be used.

|  |  |  |
| --- | --- | --- |
| **Control type** | **Command** | **Meaning** |
| VFO encoder | Up: ZZZUnn;  Down: ZZZDnn; | nn = number of clicks (0-99) |
| Other encoder | ZZZEnnm; | nn = encoder number:  0: unused  1-20: encoder 1-20, clockwise  51-70: encoder 1-20, anticlockwise  m=number of clicks (0-9) |
| Pushbutton (including encoder) | ZZZPnnm; | nn=button number (1-99, 0=n/a);  m=0: released; m=1: pressed; m=2: long pressed |
| Indicator | ZZZInnm; | nn=indicator number (1-99, 0=n/a);  m=0: unlit; m=1: lit |
| Send s/w Version | ZZZS;  Response ZZZSppnnmmm; | pp=product id  1: Andromeda 2: Aries 3: Ganymede  nn= hardware version  mmm= s/w version |

# Testing

## Testing in Arduino Alone

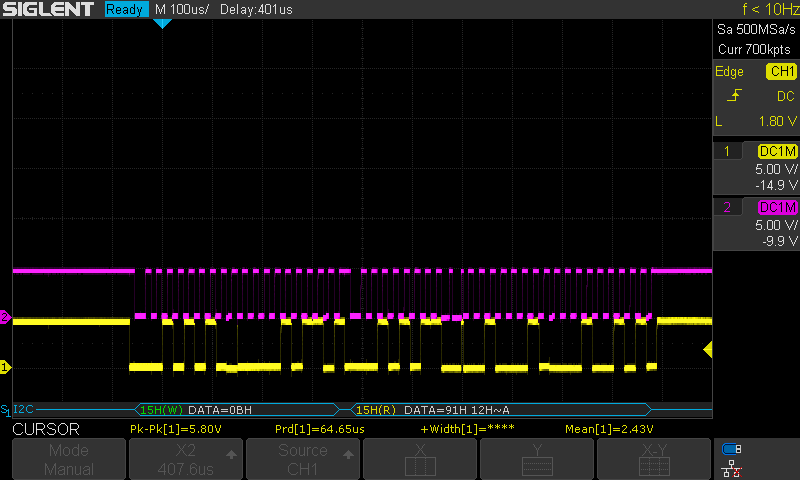
|  |  |
| --- | --- |
| Connect USB cable to PC. Open Arduino terminal window. “blink” LED blinks at 1Hz rate | Works OK |
| LED startup scan, lighting each LED in turn | OK |
| VFO encoder | c/w and ac/w turns cause events to be displayed in the serial window. Far turns get several steps displayed. |
| Dual encoders | All encoders give events for c/w and ac/w turn.  The “shifted” encoder (top right) can provide two scan codes depending on state of shift. (8&9 or 10&11) |
| Pushbuttons | All pushbuttons generate “pressed” and “released events. All give the correct scan code. “long press” declared after ~2s press, and this might be better reduced.  With button band “shift” pressed we get the alternate scan codes. |

## Testing with Raspberry Pi

|  |  |
| --- | --- |
| Powers up with no bus conflicts | OK |
| I2C discovery program can find the Arduino:  sudo apt-get install i2c-tools  i2cdetect -y 1 | OK; finds correct address |
| Data can be pushed to Arduino | Can set LEDs |
| Data can be read from Arduino | Can read panel s/w version, encoder and pushbutton data |
| Interrupt asserted if data in queue | Yes |
| Operational data transfer | Interrupt driven code can read I2C and read out and display all events |

## I2C communication issues

I get occasional failures to transfer a “1” bit on bit 7 of the 1st byte sent back by the Arduino. Maybe 5% of the time.



There seem to be some known issues related to clock stretching:

<https://raspberrypi.stackexchange.com/questions/91410/rpi-to-arduino-i2c-block-data-read-fails-with-errno-121-remote-i-o-error-pyt>

<https://github.com/raspberrypi/linux/issues/4884>

<https://github.com/raspberrypi/linux/issues/254>

The pragmatic thing to do is to edit the message format so that bit is not used.

# p2app Control

If the panel is controlled by p2app, then it will need to exchange “CAT over TCPIP” messages with Thetis.

Thetis operates a timeout: if a connection does not exchange data for 30s, it times out and drops the connection. “Keep alive” message exchange works, and a 15s message request means connection is maintained indefinitely.

Encoders and pushbuttons can all push data to Thetis. It will not be practical to have Thetis push data to the panel (there is no longer a specific CAT instance to be used) and the panel will need to use CAT commands to establish the LED state.

|  |  |  |
| --- | --- | --- |
| **s/w number** | **Function** | **CAT message** |
| 1 | MOX | Combined VFO Status |
| 2 | Tune | Combined VFO Status |
| 3 | 2 Tone | 2 Tone (ZZUT) |
| 4 | ATU Tune solution | (internal to panel) |
| 5 | ATU enabled | (internal to panel) |
| 6 | XIT | Combined VFO Status |
| 7 | RIT | Combined VFO Status |
| 8 | A/B | A/B (ZZYR) |
| 9 | VFO lock | Combined VFO Status |
| 10 | Band shift | (internal to panel) |
| 11 | Encoder shift | (internal to panel) |

|  |  |  |
| --- | --- | --- |
| **Message** | **CAT message detail** | **Notes** |
| Combined VFO Status | Get: ZZXV;  Ans: ZZXVNNNN;  NNN = 0 - 1023 | Combines reporting of RIT, LOCK, SPLIT, CTUNE, MOX and TUNE status  Bit 0: RIT on/off (see ZZRT)  Bit 1: VFO A LOCK status (see ZZUX)  Bit 2: VFO B LOCK status (see ZZUY)  Bit 3: SPLIT status (see ZZSP)  Bit 4: VFO A CTUNE status (see ZZCN)  Bit 5: VFO B CTUNE status (see ZZCO)  Bit 6: MOX status (see ZZTX)  Bit 7: TUNE status (see ZZTU)  Bit 8: XIT on/off (see ZZXS)  Bit 9: VFO SYNC on/off (see ZZSY) |
| 2 Tone on/off | Get: ZZUT;  Ans: ZZUTN; | N=0: off;  N=1: 2 tone test active |
| A/B collapsed display buttons | Get: ZZYR;  Ans: ZZYRN; | N=0: RX1;  N=1: RX2 |

# Arduino Software Installation

This guide describes how to download, install and load the Arduino software for the Odin console. The guide assumes that you are using the Arduino Integrated Development Environment (IDE) running on a windows platform. For users with different operating systems, different folder locations will probably apply.

## Install the Arduino IDE

The Arduino IDE is downloaded from the Arduino web page. The download links are on this page:

<https://www.arduino.cc/en/Main/Software>

Download and install the IDE. When you run it for the first time, it will look something like:



This is showing you a new, blank program. Arduino programs are called “sketches”.

## Add Support for the Nano Every Boards

As shipped the Arduino IDE can build code for some of the processor types used in the Arduino range, but not for the Arduino “Due” used in this project. A simple download will add the Due:

1. Open the Arduino IDE
2. Click “Tools|Board|Boards manager” on the menu
3. Scroll down to the entry for “Arduino Mega AVR boards by Arduino” and click “install”
4. Your screen should look something like this:



## Download the Panel Software Repository

1. Visit the repository on github: https://github.com/laurencebarker/SaturnG2V2\_Front\_Panel
2. Click “clone or download” then “download zip”
3. Store the zip file on your PC for example in the “downloads” folder
4. Open the zip file and extract to your PC; for example into a folder “SDR” in “documents”
5. There will be a folder called “Andromeda\_front\_panel-master” in your “SDR” folder

There are several folders:

|  |  |
| --- | --- |
| documentation | The user guide and this installation guide |
| g2v2panel | Folder for the Arduino sketch |
| hardware | h/w design schematics etc |
| pipaneltest | Testing data |

## Build the code

To open the appropriate software sketch (the filenames etc are listed in the tables above)

1. Run the Arduino IDE
2. Use the "File|Open..." menu command
3. Open the “g2v2panel” folder
4. Navigate to " g2v2panel.ino" and click "open"
5. you should now see the files listed in tabs above the editor window

You now need to tell the IDE what kind of board it is compiling for, and which serial port to use to connect to it.

1. Connect a USB cable between the Arduino programming port (next to the black power connector) and your PC.
2. It may be necessary to install device drivers at this point – follow any instructions.
   1. Click "board" on the "tools" menu and select “Arduino Nano Every”
3. Select “register emulations” on the “Tools” menu to say “none (ATMEGA4809)”
4. Click “port” on the “tools” menu and choose the Arduino COM port listed (mine is COM6)
5. Click "Verify/compile" on the "sketch" menu to compile
6. (A message “compiling sketch…” will appear. This will take around a minute and should result in a message saying the % of program space used)

Finally you need to upload the code to your Arduino:

* Click "Upload" on the "sketch" menu to upload to the Arduino
* A simple progress bar will show in the bottom window of the IDE, twice - for each of "programming" and "verify"
* When it has successful finished the last message will be "CPU reset"
* (note that an error message **avrdude: jtagmkII\_initialize(): Cannot locate “flash” and “boot” memories in description** is reported for the Arduino Nano Every but this can be ignored)

Your Arduino should now be executing the Andromeda code!