# FreeRTOS DEV API - Extension Interface and MOS Analysis

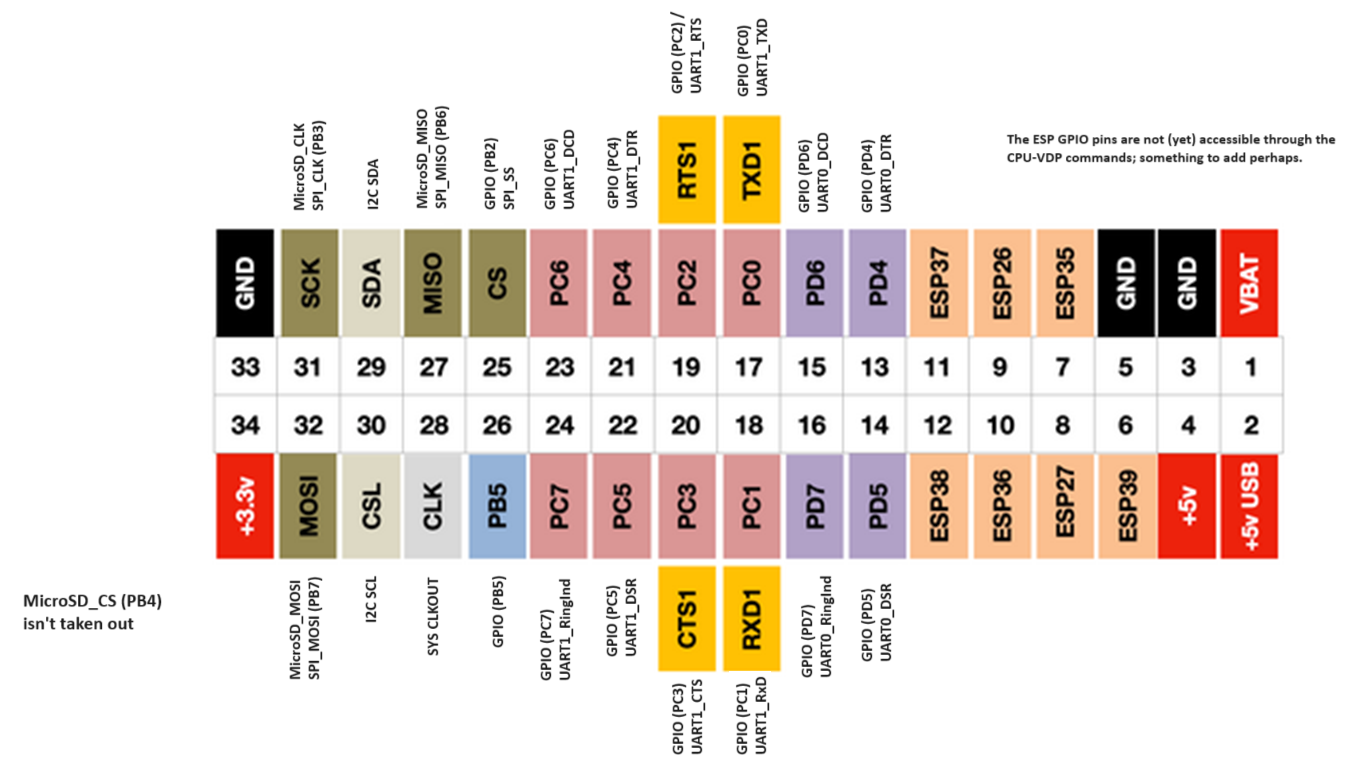
Julian Rose

## Summary

The goal of the FreeRTOS DEV API is to develop a uniform method for programmers to access the Agon Light (and compatibles) hardware Extension Interface, while using as much as possible of the existing ROM-based MOS services[[1]](#footnote-1). The hardware interfaces accessible are I2C, GPIO, SPI and UART. The Agon hardware platforms have no issues, and fully support these interfaces as per the eZ80 capability. However, the MOS software does have issues and use-case limitations, common to both the current Quark (MOS 1.04) and Console8 (MOS 2.x.y) software, that need addressing to meet our goal.

## Extension Interface

The Agon Light, Origins, and Olimex Light 2 (depicted) Extensions Interface are compatible for DEV API.



Refer to <https://agonconsole8.github.io/agon-docs/GPIO/>. The DEV API shall address functions at PD4-PD7, PC0-PC7, CS[[2]](#footnote-2), PB5, MISO, SDA, CSL, SCK, and MOSI. DEV API shall not address any of the remaining pin functions. Platform pinout differences are outlined: Agon Origins 34-pin pin 1 (VBAT) is not connected, but is otherwise an alike pinout; Agon Light has a 32-pin connector, without VBAT and 5v outputs[[3]](#footnote-3), but is otherwise alike; the Console8 40-pin pinout differs more substantially; but all platforms are sufficiently compatible such that just one version of the FreeRTOS DEV API software will run on all of them.

## DEV API outline and MOS Issues

We outline designs for the FreeRTOS DEV API and highlight issues with the MOS software for each of the functional groups in turn. Each functional group (I2C, GPIO, SPI and UART) shall have a unique major number identifying it in the DEV API software. Individual devices shall each have a unique minor number within their major number grouping (like \*nix).

### GPIO

The MOS API does not expose any GPIO functions. With safeguarding configured, and on application call to Open a device, the FreeRTOS DEV API software shall check that the requested pin(s) have not been previously allocated prior to successfully allocating the GPIO function. Each GPIO pin may be opened as an independent device, with the minor number distinguishing it. Minor numbers shall be assigned as per the pin numbers; for example, the device at PB5 (pin 26) shall be GPIO minor device 26. Note that GPIO 25, PB2 (alternately SPI /SS) cannot be assigned to GPIO because of issues in the MOS SPI software (see below). The remaining GPIO-alternate pins (13..16, 17..24, and 26 – that is 13 in total) can be assigned.

#### MOS GPIO Issues

The hardware reset default for all GPIO pins is Mode 2 = Input with interrupts disabled. The MOS file src/gpio.asm contains one public function, GPIOB\_SETMODE to set the mode of port B. This is called once from src\_startup/init\_params\_f92.asm (modified version of the Zilog startup), to configure Port B2 (GPIO 25 / SPI SS) as Mode 9 (as per PS015317 table 6, a Rising Edge interrupt input). This may be a left-over from testing MOS SPI. Note also the bug in src/gpio.asm:GPIOB\_SETMODE::GPIOB\_M4 dual edge interrupt, should be clear DDR, clear ALT1, set ALT2; although that mode is unused within MOS.

#### DEV API GPIO

There is no MOS API for programmers to invoke GPIO functions. The FreeRTOS DEV API shall craft new software. There are numerous examples of setting up GPIO registers in the MOS software to build on.

#### GPIO Errata – Edge-driven interrupts

Hardware Bug: Refer to Zilog UP0049 item no.6, “GPIO edge-trigger interrupt mapping error”. Use of *edge-driven* interrupts (modes 6 and 9) are limited due to a bug in the chip logic. In particular, interrupt clear down will not work correctly for certain combinations of GPIO and alternate function pin assignment within any given port.

Resolution: To achieve proper interrupt clear down behaviour for a particular port pin, its mirror pin must be programmed in a similar manner (i.e. edge-driven interrupt). For any port, for any pin, its mirror pin = 7 - pin#; giving the mirror pin pairs {(7,0),(6,1),(5,2),(4,3)}.

Impact for Agon Light2 at Extension Interface:

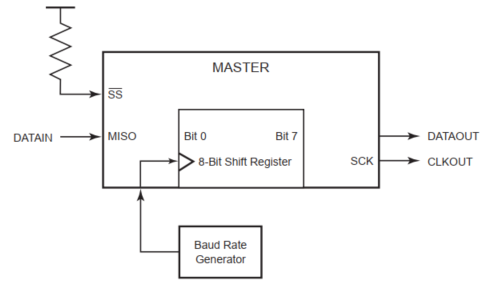
|  |  |  |
| --- | --- | --- |
| Port / Pin | Mirror → Function | Notes |
| PD7 | PD0 → Uart0 TX | Tested using PD7 in DEV\_MODE\_GPIO\_INTRDE with PD4 DEV\_MODE\_GPIO\_OUT. PD7 falling edge events are captured and cleared. PD7 rising edge events are not captured. No resolution as UART0 is the eZ80-VDP link. |
| PD6 | PD1 → Uart0 RX | Tested similar to PD7. PD6 rising edge events are not captured. PD6 falling edge detections only. |
| PD5 | PD2 → Uart0 RTS | Tested similar to PD7. PD5 rising edge events are not captured. PD5 falling edge detections only. |
| PD4 | PD3 → Uart0 CTS | Tested using PD4 configured as DEV\_MODE\_GPIO\_INTRDE with PD5 DEV\_MODE\_GPIO\_OUT. PD4 Rising edge events are not captured. PD4 Falling edge detections only. |
| PC7 | PC0 → Uart1 TX | Similar to PD7 iff UART1 is used simultaneously in DEV\_MODE\_UART\_MODEM\_FLOWCTRL. Tested using PC7 in DEV\_MODE\_GPIO\_INTRDE with PD4 DEV\_MODE\_GPIO\_OUT. Both PC7 rising and falling are captured. |
| PC6 | PC1 → Uart1 RX | Similar to PD6 iff UART1 is used simultaneously in DEV\_MODE\_UART\_MODEM\_FLOWCTRL; should be okay otherwise. Not yet tested. |
| PC5 | PC2 → Uart1 RTS | Similar to PD5 iff UART1 is used simultaneously in DEV\_MODE\_UART\_MODEM\_FLOWCTRL; should be okay otherwise. Not yet tested. |
| PC4 | PC3 → Uart1 CTS | Similar to PD4 iff UART1 is used simultaneously in DEV\_MODE\_UART\_MODEM\_FLOWCTRL; should be okay otherwise. Not yet tested. |
| PC3 | PC4 → Uart1 DTR | Similar to PD3 iff UART1 is used simultaneously in DEV\_MODE\_UART\_HW\_FLOWCTRL; should be okay otherwise. Not yet tested. |
| PC2 | PC5 → Uart1 DSR | Similar to PD2 iff UART1 is used simultaneously in DEV\_MODE\_UART\_HW\_FLOWCTRL; should be okay otherwise. Not yet tested. |
| PC1 | PC6 → Uart1 DCD | Similar to PD1 iff UART1 is used simultaneously; should be okay otherwise. Both PC7 rising and falling are captured. |
| PC0 | PC7 → Uart1 RI | Similar to PD0 iff UART1 is used simultaneously; should be okay otherwise. Not yet tested. |
| PB5 | PB2 → SPI SS | As PB2 is a constant level, it will impact interrupts on PB5. Tested using PB5 configured as DEV\_MODE\_GPIO\_INTRDE with PD4 DEV\_MODE\_GPIO\_OUT. PB5 rising edge events are not captured. PB5 falling edge detections only. |
| PB2 | PB5 → GPIO | We can’t use PB2 (pin 25) for GPIO due to MOS SPI software issues |

### SPI

The MOS API does not expose any SPI functions. On application invocation, the FreeRTOS DEV API shall check that a requested (/SS emulation – see below) pin is not allocated prior to successfully allocating the SPI device. Each SPI slave device shall be uniquely distinguished with a minor number. Minor numbers shall be as per the pin number assigned as /SS; for example, device with /SS assigned to PB5 shall be SPI device 26.

#### MOS SPI Issues

MOS main.c calls src/spi.asm::\_init\_spi to setup the SD-card interface. The comment “SS must remain high for SPI to work properly” is clarified with reference to Zilog PS015317 fig 29 eZ80 SPI as a Master Device:



SPI designates a single fixed Master device in a network (the role of Master is not transferable within the SPI standard). This device is responsible for generating the clock signal, SCK, in all data exchanges. For the Agon Extension Interface, PB2 fixes the SPI Master[[4]](#footnote-4); any attached devices (including the SD-card device) are SPI Slaves.

MOS \_init\_spi re-configures PB2 to GPIO Mode 1 (as per PS015317 table 6, a GPIO-controlled constant ‘1’ output). (It then does similar for PB4, which is the SD-card /CS, to de-select the SD-card.) This PB2 configuration fixes the eZ80 as the SPI master. (As the MOS 1.04 software stands, PB2 will output a continuous ‘1’ value once SPI is initialised. We can observe this with a scope or multimeter.)

The eZ80 SPI requires an 8-bit value to be written to SPI\_TSR to initiate a SPI I/O operation. MOS writes the constant 0xFF in spi.asm::spi\_read for example, which is a dummy byte for the SD-card. In the general case DEV API shall program the output value.

#### DEV API SPI

With PB2/SS fixed high, DEV API permits other GPIO pins (e.g. pin 26, PB5) to emulate /SS (normally high, asserted low). Each emulated /SS pin shall be designated with a unique minor number as per its pin number - for example, device at PB5 shall be SPI minor device 26. Each minor number thereby gives access to a distinct SPI Slave device.

##### /SS driving control

DEV API shall assert an emulated /SS pin (driving it low) on task-invocation of an API function. And conversely, DEV API shall de-assert an emulated /SS pin (driving it back high) on completion of each function.

The SD-card will not be accessible while another SPI device is selected (which will affect the design of applications that wish to log data from SPI device to SD-card – requiring a memory buffer).

##### Slave Interrupts

Slave Interrupts (to request a transfer) are not a part of the SPI protocol. The SPI-protocol requires the application software poll the devices attached via the Extension Interface, to retrieve any data from them. However, independently of DEV API and the SPI protocol, and to facilitate slave request for a SPI data exchange, the application developer can assign a second GPIO pin (in addition to emulated /SS) to generate an interrupt. This interrupt routine could invoke an application task SPI device operation.

##### Multiplexing

In a system containing multiple SPI devices, and to make best use of the limited number of /SS emulation pins that can be allocated at the Extension Interface, the Agon user may opt to place a line decoder or fanout IC, e.g. an SN74HC138, in-circuit with external SPI device chip selects. A decoder converts N inputs into 2N outputs (a binary transformation). For example, allowing 3 /SS pins to reach 8 devices. In such an arrangement, the user application shall Open three SPI minor devices with a Group Mode attribute, so that the /SS emulation pins are asserted in a binary 2N format (rather than individually, N) when performing I/O operations.

The total number of devices that can be supported is decided by one of: the number of available GPIO lines that can be assigned to emulate /SS, including any use of Group Mode chip selects; the total electrical load that can be drawn from the whole Agon system[[5]](#footnote-5); the wire distance operating at the Agon SPI clock rate of ≈3MHz (fixed in the MOS software, presumably the maximum rate of the on-board SD-card device).

### UART

The MOS code performs well for both UART0 and UART1. It appears derived from the ‘Poll’ subset of the Zilog UART software base.

The MOS code supports either 2-pin ‘software flow control’ (although it doesn’t use Xon / Xoff control bytes, 0x17 / 0x19 respectively, so is actually ‘no flow control’) or 4-pin half-duplex ‘hardware flow control’ (RTS/CTS). Unsurprisingly, MOS doesn’t support 8-pin full-duplex ‘modem flow control’. All of these control flow modes shall be supported through the DEV UART API.

Although it enables the FIFO, the MOS code does not make use of the device FIFO buffers or interrupt-mode capability. Consequently, “poll-mode” ties up CPU-time in performing serial data transfer “byte bashing”, when it could otherwise spend some of that time multi-tasking. However, that makes the MOS code simpler, and is all that is required for the Uart0 CPU-VDP link in Agon Light; we note this also works well with the hexload utility (for uploading program binaries without the need for SD card ‘jockeying’).

With Uart0 dedicated to the CPU-VDP link, only UART1 (minor device 1) is available for FreeRTOS. With safeguarding configured, the FreeRTOS DEV API software shall check that a requested pin has not been allocated prior to successfully allocating the application-requested UART1.

The application will be able to open a UART1 connection in one of seven modes. These different modes require allocation of different pins, and support different data flow control methods. The simplest loopback mode requires no pins and does without flow control.

Four modes will support point-to-point connected peer DTEs (data terminal equipment, such as the Agon to a laptop) over short distances. These four modes differ in wiring and flow control methods. Data rates decrease with distance due to signalling capacitance and noise.

Two modes will enable connection over longer distances and require external DCE (data communication equipment[[6]](#footnote-6)). DCE-type is not specified and can include various network signal carrier technologies (wire, wireless). One of these modes supports a point-to-point connection, and the other supports addressed connections – the only mode to do so. These two modes generalise the ability of Agon to act as a micro-controller of remote end equipment. All modes enable full duplex (simultaneous tx and rx) between peer-to-peer micro-controllers.

Details of these seven modes are found in DEV UART Description referenced below.

#### MOS UART Issues

There’s a bug in src/serial.asm:UART1\_serial\_GETCH::189, such that it tests whether UART0 is enabled rather than UART1. An outcome is that if an application calls mos\_api\_ugetc when no external device is connected to UART1, then the application will busy wait evermore; needing a hardware reset.

#### DEV API UART

DEV API shall craft new software to use the eZ80 UART device with FIFO buffers in interrupt mode. It shall provide the ability to operate in one of seven modes. For more detail see the DEV API Description in the references below.

### I2C

The MOS API provides functions to support an I2C bus in Single Master roles only (akin to SPI use). And is limited to 7-bit addressing. DEV I2C shall support Multi Master roles and 10-bit addressing. With safeguarding configured, the FreeRTOS DEV API software shall check that the I2C device has not been allocated prior to successfully allocating it.

The eZ80 I2C device operates from dedicated pins (with no alternate functions). Only one I2C device is available, minor number 0.

#### MOS I2C Issues

The MOS I2C implements Single Mastering, through Master Transmit and Master Receive roles for writing and reading slave devices (sensors, actuators) respectively[[7]](#footnote-7); but omits the Slave Receive and Transmit (Slave Addressed) roles, for responding to another bus master. This limits the Agon capability as a Micro-Controller to inter-network.

Of lesser import the MOS I2C software implements 7-bit addressing only; it does not implement 10-bit extended addressing[[8]](#footnote-8). With 16 reserved 7-bit address numbers, this will be a problem only if: i/ the number of attached devices exceeds 112, which use-case seems rare – but e.g. Christmas tree LEDs; or ii/ if an application-specific sub-netting address scheme is to be used, exhausting the logical assignment of minor numbers; or iii/ if certain devices only support 10-bit addressing (e.g. on 7-bit address assigned numbers exhaustion – see <https://learn.adafruit.com/i2c-addresses/the-list> for example).

MOS does not support the I2C RESTART pattern, in which a Master Stops one transaction and immediately Starts another without relinquishing the bus. For Single Mastering this may be adequate; although certain devices may expect a RESTART in order to read register addresses. But it is inadequate for Multi-Mastering.

#### DEV API I2C

DEV I2C broadly recognise two Use Cases:

1/ Master-Controller is like the SPI use case. In this, a sole controller accesses a number of slave devices (sensors and/or actuators) on the I2C-bus.

MOS I2C supports two of the possible four roles: Master Transmit and Master Receive. (MOS omits 10-bit addressing and Restart.) These Single-Master roles are sufficient for controlling many sensors and actuators attached to the I2C bus (similar to SPI use-cases). Applications may open DEV I2C in DEV\_MODE\_I2C\_SINGLE\_MASTER mode for this purpose. Or they may continue to use MOS I2C (and disable DEV I2C to save RAM footprint) if multi-tasking is not used. In applications with multi-tasking, DEV I2C is necessary.

2/ Multi-Master is a network pattern in which a number of controllers inter-connect via the I2C bus. An Agon Cluster is such an arrangement. Simultaneously, slave devices may be present on this I2C bus, such that Master-Controller is a subset of Multi-Master with exactly one Master.

## Agon Cluster

An Agon Cluster is a system containing a number of Agon Light (or compatibles) networked together. A Cluster enables distributed, truly concurrent applications. While none of the other interfaces (UART, GPIO, SPI) lend themselves well to inter-networking a number of Agons, there are possibilities using I2C.

The I2C Multi-Master use-case supports Agon peer-to-peer networking. The ability to natively inter-connect Agon devices is limited[[9]](#footnote-9). I2C is a true multi-master bus, providing arbitration and collision detection, and presents a way of peer networking through the Extension Interface (including Light2 UEXT). With the addition of Slave-addressed read and write modes, and corresponding interrupt support, we should be able to realise an Agon Cluster.

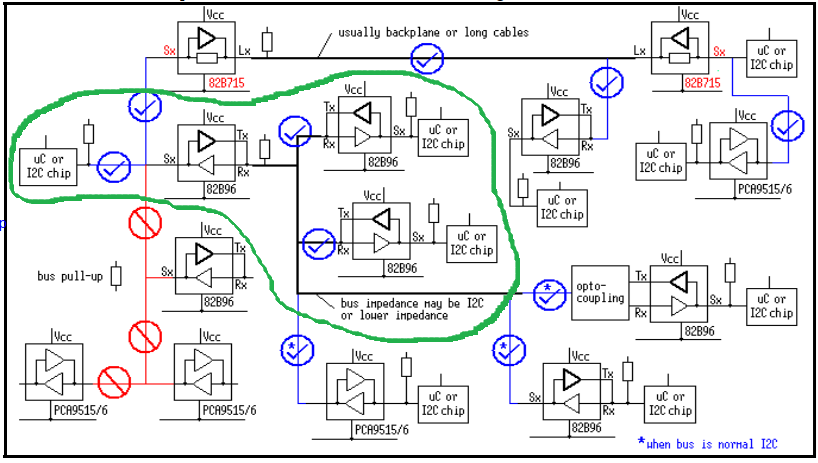
I2C specifies General Call Addressing and the Device ID protocol which can be used to determine the type of each bus-connected Device. DEV I2C shall support this, in order that multiple Agon devices can find each other.

### Backplane

Agon embeds (2K2 in the case of Light2 or 4K7 for Light) pull-up resistors on each of SCL & SDA. Connecting Agons together combines their I2C pull-ups in-parallel. When powered up, the resulting 1/R sum for N=2 Light2s gives 1K1, the minimum for an operational I2C-bus. This might function adequately for N=2 Agon Light2[[10]](#footnote-10), but will sum to be out of specification for N>2. Moreover, if any connected Agon is powered-off, its SDA & SCL resistors create loads on the circuit. This pulls the SDA & SCL lines out of specification, such that any powered Agon in such a circuit will be unable to access the bus successfully.

In general, to create an N Agon network, N P82B96 buffer ICs should be wired in parallel, such that: the Tx/Rx pins of all the P82B96 are wired to form a single common signalling backplane; and the Sx/Sy pins of each P82B96 are wired to form separate local I2C busses, to which a single Agon is attached. Each Agon will power its own local I2C bus, and the devices attached to it.

The following figure, taken from https://www.nxp.com/docs/en/application-note/AN255.pdf, is highlighted in green to show such a parallel backplane, connecting three local I2C busses each with its own uC (Agon):

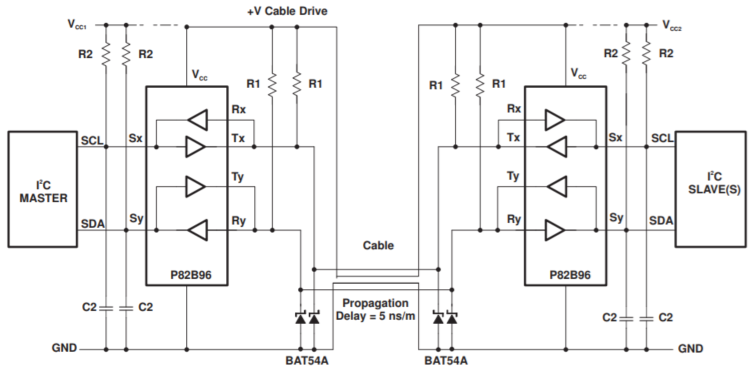


Slave devices attached to any of the local I2C busses will be visible to all Agon Masters across the backplane. This arrangement will function successfully for any powered Agon; un-powered local I2C busses and devices will not be visible, while the backplane will function.

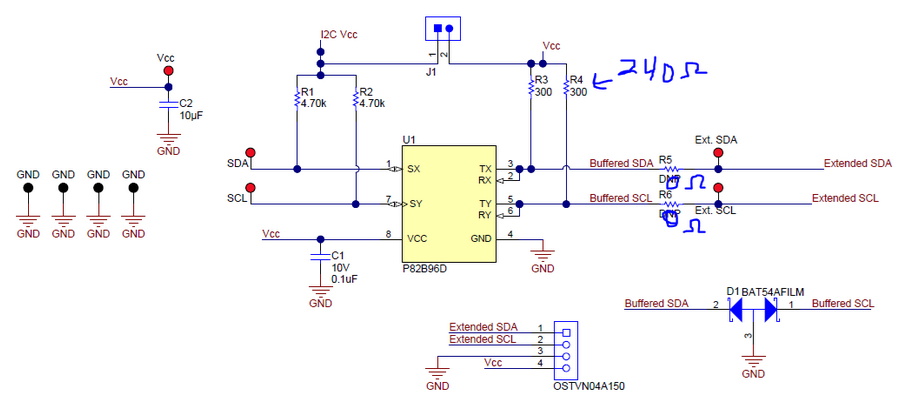
#### Testing

NOTE WE FAILED TO GET THE PEER-I2C ELECTRICAL SIGNALLING WORKING USING AGON LIGHT2 – EITHER BECAUSE OF THE I2C SDA/SCL 2K2 PULL-UPS IN PARALLEL, OR BECAUSE IN USING A P82B96 BACKPLANE THE LOGIC LOW VALUE AT 0.8V IS AT THE MAXIMUM FOR EZ80, SUCH THAT THE I2C SIGNALLING FALLS OUT OF ELECTRICAL SPECIFICATION. HOWEVER, I2C PEER NETWORKING MAY WORK WITH AGON LIGHT OR ORIGINS WHICH HAVE 4K7 PULL-UPS.

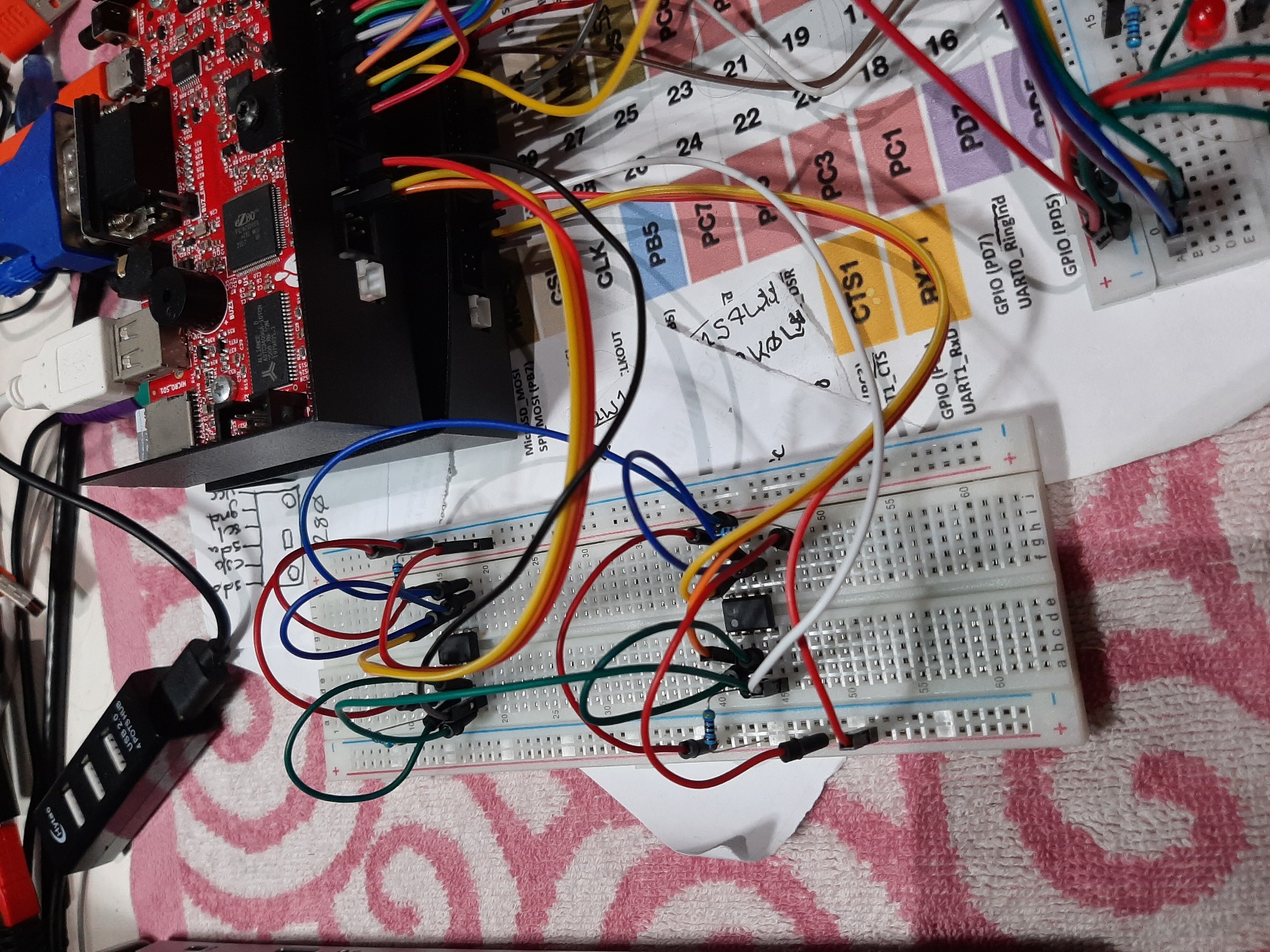
[TI-P82B96] gives us the initial circuit design:



This is refined with the design guideline:

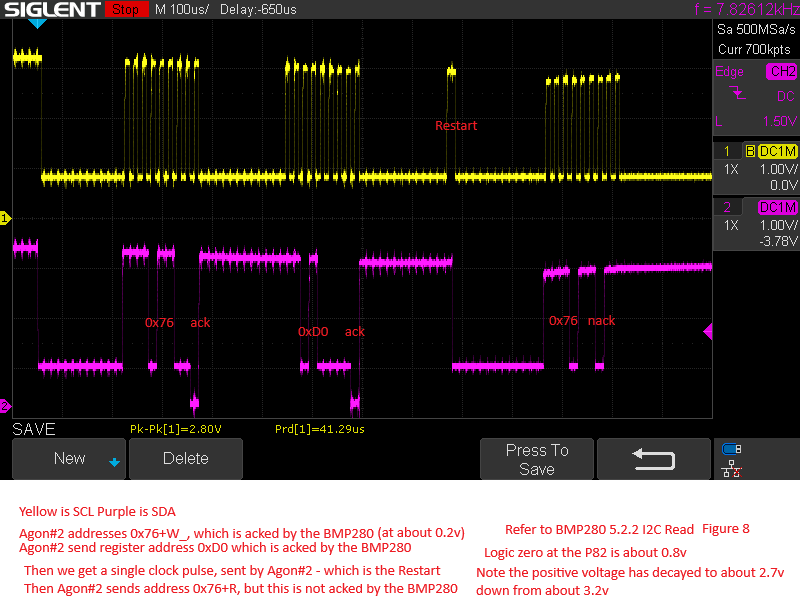


Excepting that R1 and R2 are the on-board 2k2s of our Agon Light2. With the two Light2s in parallel they sum to 1K1s. We achieved 235Ω for R3 and R4 by using 4K7s in parallel on the breadboard:



##### Barometer test

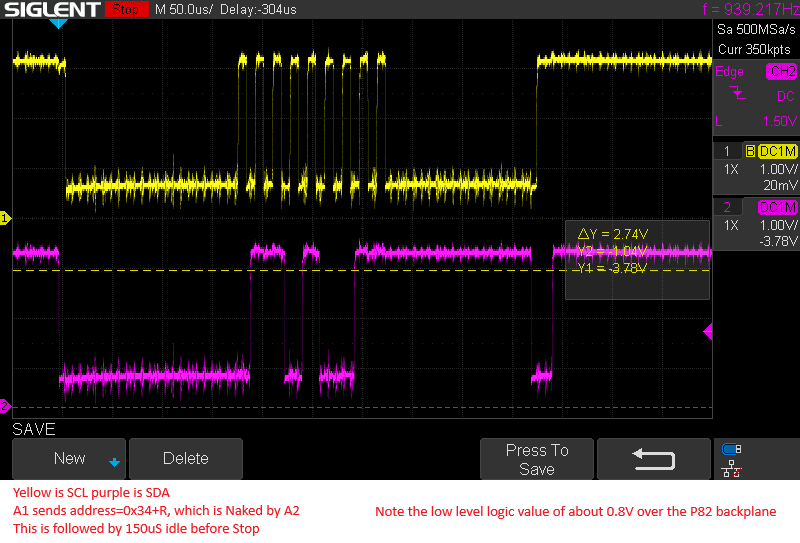
To test remote sensor access we wired a BMP280 barometer through the P82B96 backplane. The Beta test program addressing the barometer revealed signalling issues:



As seen in the above oscilloscope screen capture, the Barometer would initially ACK is address, but then fail to ACK after a restart, once the signal degrades. Voltage degradation is perhaps due to capacitance in the wires. This might improve with an expansion board rather than using a breadboard.

##### Slave addressing test

Secondly we wired two Agon Light2 devices over a P82B96 backplane through their UEXT1 pinouts. The peer-network test program shows the Slave fails to acknowledge its own address:



The eZ80 electrical characteristic permits 0.8V as the maximum voltage to be recognised as a logic Low. So, like the Light2 Pull-Ups being on the limit, it seems likely the P82B96 signal falls out of spec.

### Long-Distance

I2C was designed for intra-board connectivity, but is widely used to inter-connect devices with up to 2m of wire requiring no further technology. For longer distance connectivity, P82B96 buffers can be used, to reach up to 200m. To inter-connect Agon over even longer distances, we would need to use DEV UART with Modems or Bridges or Wifi, with an increase in cost and complexity and energy consumption.

## DEV API General Solution Options

As much as possible we want to locate software in ROM, to keep the RAM footprint to a minimum. Where we need to use RAM, that use should be user-configurable.

### 1/ ROM – modify MOS

We could fork and extend the MOS Quark software to produce a new release (maybe numbered 1.1.0 as a new and distinct branch of MOS): fix the UART1 bug; finish the I2C software with multi-mastering, to include slave-addressed modes and status register cases; fix the GPIOB\_SETMODE bug, add new GPIO code and extend the MOS API accordingly; add new SPI code, extending the MOS API. (Haven’t checked if this will fit in to the available on-chip ROM, but would expect it to.)

The latest Console8 MOS version 2.x.y can merge the changes in straight-forwardly, if their desire is to remain compatible.

There is much merit in this option. But we will have failed on ‘needs no re-FLASHing’ of the firmware to work. Maybe more than once if we introduce bugs.

### 2/ RAM – extend DEV API

We can create the DEV API as outlined above. The additional SPI software is feasible, although duplication of read and write functions is less desirable. If we omit Modem Flow Control until it is specifically requested, and at ≈250 lines of code, replacing UART code should be straight-forward and small enough. The GPIO is wholly new, but should be modest enough to make a configurable build-option acceptable. I2C will be the biggest re-implementation – but either the (Single Master) I2C MOS or (Multi-Master) DEV API methods can be user-selected through configuration.

Any bugs we introduce with this solution are easily fixed, without needing yet another candidate release version of MOS.

MOS 1.0x Quark and Console 2.x.y software can inherit bug fixes from DEV API at a later date with no impact on FreeRTOS applications.

## References

<https://agonconsole8.github.io/agon-docs/GPIO/> Outline of Agon Extension Interface (GPIO). Note error on pins 29,30 on the Agon Light pinout. Note the ending textual error, should read ‘go via a “transducer”’ and not a “transputer”.

<https://www.nxp.com/docs/en/user-guide/UM10204.pdf> I2C-bus specification and user manual

<https://learn.adafruit.com/i2c-addresses/the-list> Partial list of assigned 7-bit I2C addresses as an example. See also <https://i2cdevices.org/addresses>.

<https://www.analog.com/media/en/technical-documentation/app-notes/an-1248.pdf> SPI application note, outlining a number of potential implementation incompatibilities.

<https://www.mouser.com/datasheet/2/450/up0049-21068.pdf> Zilog eZ80F92 chip errata. (Also in ZDSII\_eZ80Acclaim!\_5.3.5\Documentation\Errata\.) Especially item no. 6 on edge-driven interrupts. And item no.7 on continuous uart Rx interrupts in a certain configuration.

<https://application-notes.digchip.com/003/3-5557.pdf> Maxim review of RS-232 UART operation.

<http://www.nullmodem.com/NullModem.htm> Null-modem wiring, for connecting two UART DTEs directly (without DCEs). See also: <https://en.wikipedia.org/wiki/Null_modem#Wiring_diagrams>

<https://en.wikipedia.org/wiki/Software_flow_control> Outline of UART software flow control using Xon / Xoff

<https://github.com/damogranlabs/serial-tool> Serial test tool I use to test DEV UART

<https://en.wikipedia.org/wiki/Hayes_AT_command_set> Hayes “AT” Modem command set

<https://github.com/julian-rose/FreeRTOS-for-Agon/blob/main/FreeRTOSv202212.01-LTS/FreeRTOS/Docs/FreeRTOS%20MOS%20DEV%20API.docx> DEV API description

<https://www.nxp.com/docs/en/application-note/AN255.pdf> NXP AN255 I2C Repeaters, Hubs and Expanders

[TI-P82B96] <https://www.ti.com/lit/ds/symlink/p82b96.pdf>

1. Using FreeRTOS ‘requires no re-FLASHing’ – you just build a MOS application, load and run it. [↑](#footnote-ref-1)
2. DEV API cannot control CS due to MOS software limitations, discussed below. [↑](#footnote-ref-2)
3. All platforms Extension Interfaces (I2C, GPIO, SPI and UART) operate at 3.3v peak voltage (wrt GND). [↑](#footnote-ref-3)
4. Not to be confused with Gerry Anderson’s Master Spy character in Supercar. [↑](#footnote-ref-4)
5. It would be far better to power external devices independently of Agon, to reduce the current draw – but many users will nonetheless take out the power rail from Agon. [↑](#footnote-ref-5)
6. Nowadays we would use wifi technology instead, with a micro-processor in the wifi device using a simple serial (uart, spi, i2c) handshake between that and our CPU. But if you want the retro experience, then RS232 it is. [↑](#footnote-ref-6)
7. Known as Single Master <https://www.i2c-bus.org/singlemaster/> [↑](#footnote-ref-7)
8. The address domains are mutually exclusive: the 7-bit address 0x1 is distinct from the 10-bit address 0x1. [↑](#footnote-ref-8)
9. An external wireless network device can be attached to an Agon (at the Extension Interface or Light2 UEXT1 port), but would require additional interfacing software and/or an intelligent controller. [↑](#footnote-ref-9)
10. It doesn’t; in practice the resulting I2C bus stops working. [↑](#footnote-ref-10)