# FreeRTOS DEV API - Extension Interface and MOS issues

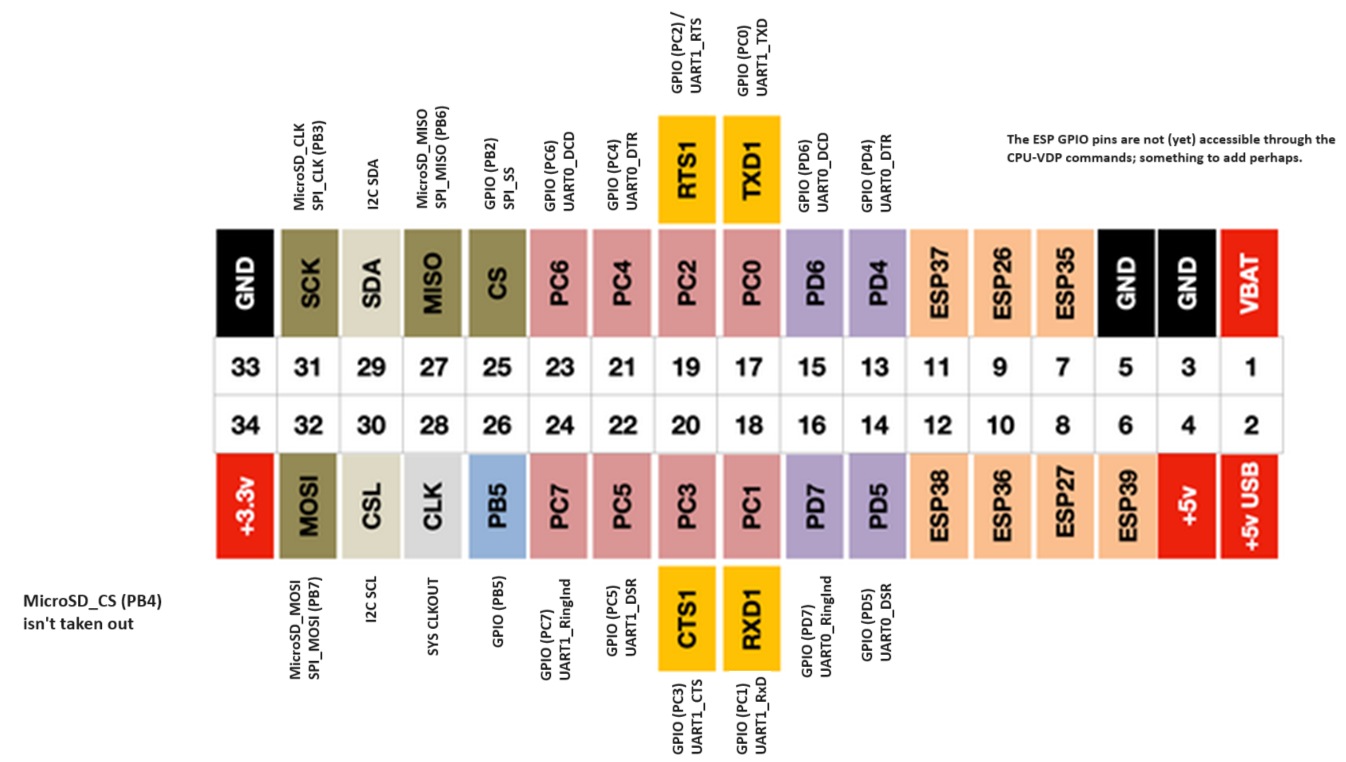
Julian Rose

## 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 significantly; 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 have a unique minor number.

### 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, device at PB5 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 13 GPIO-alternate pins (13..16, 17..24, and 26) can be assigned.

#### MOS GPIO Issues

The 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, as 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.

#### DEV API GPIO

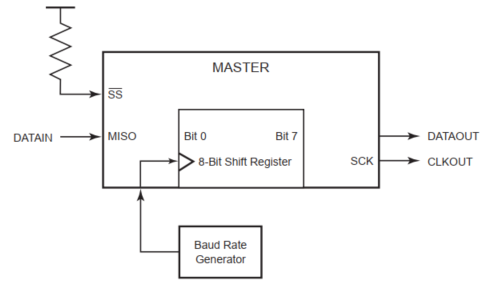
There is no MOS API for programmers to invoke GPIO functions. The FreeRTOS DEV API shall craft new software. There are numerous examples of setting up GPIO registers in the MOS software to build on.

### SPI

The MOS API does not expose any SPI functions. On application invocation, the FreeRTOS DEV API shall check that a requested (/SS emulation) 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 PS015317 fig 29 eZ80 SPI as a Master Device:



The sole SPI Master device is responsible for generating the clock signal, SCK, in all data exchanges.

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. (PB2 will output a continuous ‘1’ value once SPI is initialised.)

#### DEV API SPI

With PB2/SS fixed high, DEV API shall allocate 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 SPI device may be opened individually.

##### /SS driving control

PB4 (SD-card /SS) is normally asserted. To ensure SD-card use does not conflict with external SPI device(s) reading and writing, DEV API shall de-assert the SD-card /SS PB4 (drive it high) prior to asserting an emulated /SS pin (driving it low). And conversely, DEV API shall de-assert an emulated /SS pin (driving it back high) on completion of each I/O operation, before re-asserting the SD-card /SS PB4 (driving it low).

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 Extensions 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 to generate an interrupt. This interrupt routine could invoke a SPI device read operation.

##### Multiplexing

In a system architecture 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 a configuration, the user program shall Open three SPI 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[[4]](#footnote-4); 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).

Need to look further if src/spi.asm spi\_read and spi\_write routines can be re-used. In any case, we cannot call the spi routines (without a MOS API), so we’ll have to implement new software in DEV API.

### UART

The MOS API provides functions to access UART1, with no flow control, or with hardware flow control, but not with modem flow control. 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 UART function.

Only one UART device is available, minor number 1. (UART 0 is permanently assigned to the CPU-VDP link.) UART1 can be opened in one of three modes, that require different pins to be allocated: NO\_HW\_FLOWCONTROL, which requires pins 17 and 18 only; HW\_FLOWCONTROL, which requires pins 19 and 20 in addition; and MODEM\_FLOWCONTROL, which further requires pins 21 through 24.

#### MOS UART Issues

There’s the 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 you call mos\_api\_ugetc when no external device is connected to UART1, then the application will busy wait evermore and a system reset is necessary.

#### DEV API UART

To overcome this bug, DEV API will have to re-implement the code for UART1 (uart.c, about 100 lines of C; and serial.asm, about 150 lines) – not actually that much.

The MOS UART software doesn’t support Modem Flow Control. We would need to add this in the FreeRTOS DEV API software. We could add this at a later date, if it were requested.

### I2C

The MOS API provides functions to support an I2C bus in Single Master mode only (akin to SPI single mastering). With safeguarding configured, the FreeRTOS DEV API software shall check that the 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 modes for writing and reading devices respectively[[5]](#footnote-5); but omits the Slave Receive and Transmit (Slave Addressed) modes, for responding to another bus master. This limits the Agon capability as a Micro-Controller to inter-network.

The MOS I2C software implements 7-bit addressing only; it does not implement 10-bit extended addressing[[6]](#footnote-6). With 16 reserved 7-bit address numbers, this may be a problem if: the number of attached devices exceeds 112, which use-case seems rare – but e.g. Christmas tree LEDs; or if an application-specific sub-netting address scheme is to be used, exhausting the logical assignment of minor numbers; or 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).

#### DEV API I2C

One use-case for I2C Slave modes (bus multi-mastering) is Agon peer-to-peer networking. Presently, the ability to natively inter-connecting Agon devices is limited[[7]](#footnote-7). I2C is a true multi-master bus, providing arbitration and collision detection, and presents the best way of doing that. With the addition of Slave-addressed read and write modes, and corresponding interrupt support, we might be able to realise an Agon Cluster!

## 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.

### ROM – modify MOS

We could fork and extend the MOS Quark software to produce a new release, 1.05 (or maybe 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.

The latest Console8 MOS version can merge the changes in straight-forwardly.

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. And personally, I don’t have a working Acclaim! Smart Cable hardware debugger yet (device driver problem); which will probably be needed.

### 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. The MOS software can inherit the 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 (so-called 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.

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. 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-4)
5. Known as Single Master <https://www.i2c-bus.org/singlemaster/> [↑](#footnote-ref-5)
6. The address domains are mutually exclusive: the 7-bit address 0x1 is distinct from the 10-bit address 0x1 (although a single device could respond to either). [↑](#footnote-ref-6)
7. Something like an external wireless network device can be attached. [↑](#footnote-ref-7)