# Application Programming Interfaces for FreeRTOS / MOS for Agon Light

Julian Rose

We primarily describe the Application Programming Interface (APIs) of FreeRTOS / MOS for the Agon Light and compatibles. These APIs enable user application programs to access the system services and various capabilities in the C programming language. We assume the reader is acquainted with Agon Light hardware [Agon], the MOS software [MOS], FreeRTOS, and the C programming language in general, so we do not describe them further herein. See the References for links to these subjects and more.

## Summary

We briefly summarise the FreeRTOS port to the eZ80-based Agon Light.

Targeting Agon as a Micro-controller, FreeRTOS port for the Zilog eZ80-based Agon Light (and compatibles) running MOS. The concept is very much FreeRTOS over MOS, reflected in the project name. FreeRTOS provides concurrency and time, and MOS provides the system services.

This Agon Light port integrates FreeRTOS version 20221201-LTS (10.5.1) with the Zilog ZDSII eZ80Acclaim! version 5.3.5 (Build 23020901) C language toolset. The choice of version 20221201-LTS prioritises stability over latest and greatest.

### MOS versions

FreeRTOS for Agon is built to MOS version 1.04. Version 1.03 may work, but has not been tested. Likewise newer Console8 versions of MOS 2.x will work, but have not been tested. Console8-specific MOS and VDP functions are not yet supported. There are a small number of tightly coupled dependencies in parts of the code (such as the keyboard read functions) which may fail if differences start to emerge between forks of the MOS code.

### What is FreeRTOS?

FreeRTOS is a real-time software development kernel and library. It provides an application development framework that allows a C language application to be arranged into a number of concurrent tasks. The core of FreeRTOS (and indeed any RTOS) is its kernel; refer to [FreeRTOS Kernel]. And to [FreeRTOS API] for the API.

FreeRTOS is not an operating system in the sense of MOS or CP/M, which provide a command console interpreter and long-term storage. Instead, FreeRTOS for Agon applications run on MOS to access its services. No re-Flashing is required - you just build and link a C language application together with the FreeRTOS software into a MOS executable in the usual way.

#### Why do we care about concurrency and time?

In a nutshell, because they are manifest in the real world. A micro-controller interacts with the real world, through sensors and actuators. To perform its functions well, we need to embrace time and the order of events (concurrency) into our embedded software.

## Capabilities

FreeRTOS / MOS for Agon Light is a highly configurable, multi-capability project.

|  |  |
| --- | --- |
| Capability | API |
| Alpha | FreeRTOS API, with a minimal MOS API, in eZ80 ADL mode |
| Beta | MOS API (MOS 1.04 subset of the FFS API), plus a DEV API, the hardware interfaces APIss |
| Gamma | VDP API (MOS 1.04 subset), the graphics API |
| Delta | Console8 MOS 2.x & VDP extended API |
| Epsilon | A real-time API inspired by posix-4 |
| Omega | A safer version with protected Z80-mode tasks and the ADL-mode kernel |

Each capability is configurable through user-settable definitions in application-specific config header files. The different capabilities are not mutually exclusive; a user-application can be configured to include the exact mix of capabilities it requires. In this way the built executable occupies the least RAM footprint possible.

### Alpha Capability API

The Alpha capability supports all the FreeRTOS functions; the Zilog Standard C library functions; and a minimal number of MOS functions.

|  |  |
| --- | --- |
| Header Files | Functions |
| <cio.h> | getch, putch – implemented in MOS/init.asm |
| <assert.h>, <ctype.h>, <errno.h>, <float.h>, <format.h>, <limits.h>, <math.h>, <setjmp.h>, <stdarg.h>, <stddef.h>, <stdio.h>, <stdlib.h>, <string.h> | Standard C library from Zilog, including printf, memcpy, strlen, etc. |
| <atomic.h>, <croutine.h>, <event\_groups.h>, <FreeRTOS.h>, <list.h>, <message\_buffer.h>, <portable.h>, <queue.h>, <semphr.h>, <stdint.h>, <stream\_buffer.h>, <task.h>, <timers.h> | FreeRTOS library, including xTaskCreate, vSemaphoreCreate, xEventGroupCreate, xMessageBufferCreate, etc. |
| FreeRTOSConfig.h | Application Alpha FreeRTOS configuration.  configUSE\_PREEMPTION  configTICK\_RATE\_HZ  configCHECK\_FOR\_STACK\_OVERFLOW, etc. |

The Zilog header files are found under …/ZDSII\_eZ80Acclaim!\_5.3.5/include/ (where … means your installation path).

The FreeRTOS header files are found under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Source/include/.

Each application will maintain and include its own FreeRTOSConfig.h header file, to bring in exactly the FreeRTOS functions it requires and no more. See for example the Demos under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Demo/Alpha/.

### Beta Capability

The Beta capability adds support for most of the MOS API defined in [MOS API] (including the FFS API), plus an additional DEV API.

#### MOS API

|  |  |
| --- | --- |
| Header Files | Functions |
| mosapi.h | mos\_getkey, mos\_load, mos\_save, mos\_cd, mos\_del, mos\_ren, mos\_mkdir, mos\_getsysvars, mos\_copy, |
|  | mos\_fopen, mos\_fclose, mos\_fgetc, mos\_fputc, mos\_feof |
|  | mos\_getfil, mos\_fread, mos\_fwrite, mos\_flseek |
|  | mos\_geterror (plus mos\_printerr) |
|  | mos\_getrtc, mos\_setrtc, mos\_setintvector, mos\_uopen, mos\_uclose, mos\_ugetc, mos\_uputc, mos\_setkbvector, mos\_getkbmap, mos\_i2c\_open, mos\_i2c\_close, mos\_i2c\_write, mos\_i2c\_read |
| ffsapi.h | ffs\_fopen, ffs\_fclose, ffs\_fread, ffs\_fwrite, ffs\_flseek, ffs\_feof, ffs\_fstat |
| mosConfig.inc | Application Beta MOS API configuration:  configUSE\_MOS\_FILE\_OPS  configUSE\_MOS\_DIR\_OPS  configUSE\_MOS\_BUFFERED\_FILE\_OPS  configUSE\_MOS\_SYSVARS  configUSE\_FFS\_FILE\_OPS  configUSE\_FFS\_DIR\_OPS  configUSE\_MOS\_KEYBOARD\_OPS  configUSE\_MOS\_DEVICE\_RTC  configUSE\_MOS\_DEVICE\_UART  configUSE\_MOS\_DEVICE\_I2C |

The MOS header files mosapi.h and ffsapi.h are found under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Source/mos/ (where … means your installation path).

Each application will maintain and include its own mosConfig.inc header file, to bring in exactly the MOS functions it requires and no more. See for example the Demos under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Demo/Beta/. Note this is an assembly header file as the MOS API is implemented mainly in assembler.

#### DEV API

The DEV API extends the Beta capability with access to the Agon Extensions Interface directly. It provides a safeguarded API for SPI, UART, I2C and GPIO; and through them to connected devices. The DEV API enhances the Uart and I2C capability of MOS, and provides APIs for GPIO and SPI that are absent in the MOS API.

|  |  |
| --- | --- |
| Header Files | Functions |
| devapi.h | uart\_open, uart\_close, uart\_getch, uart\_read, uart\_read\_buffered, uart\_putch, uart\_write, uart\_write\_buffered, uart\_poll, uart\_ioctl |
|  | i2c\_open, i2c\_close, i2c\_readm, i2c\_writem, i2c\_reads, i2c\_writes, i2c\_ioctl |
|  | spi\_open, spi\_close, spi\_read, spi\_write |
|  | gpio\_open, gpio\_close, gpio\_read, gpio\_write |
| devConfig.h | Application Beta DEV API configuration:  configUSE\_DRV\_UART  configUSE\_DRV\_I2C  configUSE\_DRV\_SPI  configUSE\_DRV\_GPIO |
|  | configTRACE |
|  | configUSE\_DEV\_SAFEGUARDS  configUSE\_FAST\_INTERRUPTS |
|  | configDRV\_UART\_BUFFER\_SZ  configDRV\_UART\_UNBUFFERED\_DELAY |
|  | configDRV\_I2C\_BUFFER\_NUM  configDRV\_I2C\_BUFFER\_SZ  configDRV\_I2C\_TRANSITION\_RETRY  configDRV\_I2C\_MAX\_DELAY |

The DEV header file devapi.h is found under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Source/mos/ (where … means your installation path). (Private header files for each of the major device types are also found there, which are not intended for direct inclusion by application programs.)

Each application will maintain and include its own devConfig.inc header file, to bring in exactly the DEV functions it requires and no more. See for example the Demos under …/FreeRTOS-for-Agon/FreeRTOSv202212.01-LTS/FreeRTOS/Demo/Beta/.

##### DEV GPIO Description

Agon makes accessible a total of 13 GPIO pins directly from the eZ80 CPU, each independent of the other. (There are other GPIO pins accessible from the VDP ESP32, but this FreeRTOS port does not support those at this time.)

Each pin may be opened in one of several modes, corresponding to the eZ80 configuration. Namely:

|  |  |
| --- | --- |
| Mode | Summary |
| DEV\_MODE\_GPIO\_OUT | Mode 1 - standard digital output |
| DEV\_MODE\_GPIO\_IN | Mode 2 - standard digital input |
| DEV\_MODE\_GPIO\_DIO | Mode 3 - Open Drain I/O (needs external pullup) |
| DEV\_MODE\_GPIO\_SIO | Mode 4 - Open Source I/O (needs external pulldown) |
| DEV\_MODE\_GPIO\_INTRDE | Mode 6 - Input Dual-Edge triggered INTR |
| DEV\_MODE\_GPIO\_ALTFUNC | Mode 7 - Alternate hardware function |
| DEV\_MODE\_GPIO\_INTRLOW | Mode 8 - Input Interrupt active level low |
| DEV\_MODE\_GPIO\_INTRHIGH | Mode 8 - Input Interrupt active level high |
| DEV\_MODE\_GPIO\_INTRFALL | Mode 9 - Input Interrupt active falling edge |
| DEV\_MODE\_GPIO\_INTRRISE | Mode 9 - Input Interrupt active rising edge |

##### DEV I2C Description

Agon makes one I2C port available from the eZ80 CPU. MOS supports two of the possible four I2C roles: Master Transmit and Master Receive. These Single-Master roles are sufficient for controlling sensors and actuators attached to the I2C bus (similar to SPI use-cases). DEV I2C shall support all four roles, including Slave Transmit and Slave Receive, allowing other devices to act as a remote bus Master.

###### Use Cases

DEV I2C broadly recognises two Use Cases:

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 implements a subset of this Use Case: 7-bit addressing, and no restart - sufficient for most typical uses.)

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 (see [AgonCluster]). There may also be slave devices on this I2C bus, such that Master-Controller is a subset of Multi-Master, with one Master.

###### Modes

Applications may open I2C with DEV\_MODE\_I2C\_SINGLE\_MASTER mode to access the two Master roles. Or they may continue to use MOS I2C (and disable DEV I2C to save RAM footprint).

Applications may open I2C with DEV\_MODE\_I2C\_MULTI\_MASTER mode to access all four roles of Multi-Mastering. This supports networking multiple intelligent devices on the bus.

|  |  |
| --- | --- |
| Mode | Summary |
| DEV\_MODE\_I2C\_SINGLE\_MASTER | Agon as a single I2C Bus Controller |
| DEV\_MODE\_I2C\_MULTI\_MASTER | Agon as one of multiple I2C Bus Controllers |

Refer to [AgonCluster] for one possible application of DEV\_MODE\_I2C\_MULTI\_MASTER.

###### Transactions

Transactions work well for DEV UART because the UART device has a hardware FIFO (of depth 16). Each interrupt allows up to 16 bytes to be transceived without CPU involvement. Enough time for a concurrent task to do useful work. Hence, for the UART (DEV\_MODE\_BUFFERED) non-blocking transactions are useful.

Whereas, the eZ80 I2C device (like its SPI device) does not have a FIFO; such that each transceived byte will be reported to the CPU by an interrupt, causing a context switch from the current task to the ISR each time. This interrupt frequency leaves less useful time for a concurrent task to run. Moreover, unlike DEV UART in which reading and writing are asynchronous and can be full duplex with concurrent tasks (using DEV UART), access to the I2C-bus is limited to one active transaction at any time.

However, transactions are a useful way to implement ISR-driven state transitions - with the added benefit of minimal user-application involvement - especially in support of I2C bus multi-mastering (Slave Receive and Slave Transmit roles). Therefore, DEV I2C shall use Blocking Transactions, such that DEV I2C-invoking tasks will block for the duration of each call; while Non-blocking transactions shall be omitted. All transactions will execute in the context of i2cisr using transaction buffers, similar to UART.

###### Master Transmit and Receive role handling

Master Transmit and Master Receive roles shall be invoked by application calls to i2c\_writem and i2c\_readm respecitvely. DEV I2C shall not provide buffers for these functions, as the task-provided parameters are sufficient, and avoid the need for memcpy.

Calling i2c\_open with parameter DEV\_MODE\_I2C\_MASTER will access DEV I2C in Master role only (no Slave role).

###### Slave Receive role handling (including Global Call Address)

Calling i2c\_open with parameter DEV\_MODE\_I2C\_MASTER\_SLAVE will access DEV I2C for both Master and Slave roles.

In DEV\_MODE\_I2C\_MASTER\_SLAVE mode DEV I2C shall respond automatically to requests for Device ID, and shall support General Call Addressing provided the application has enabled GCA (by calling i2c\_ioctl).

Slave Receive role is the activity of receiving data from a remote Master. DEV I2C shall manage Slave Receive (including GCA) automatically by using local buffers. The application shall subsequently retrieve Slave Receive data by calling i2c\_reads, and needs to do so in a timeframe that avoids local buffer overrun if too slow. The rate of Slave Receive data arrival is application-specific.

DEV I2C supports the Global Call Address, but it is not enabled by default. To enable GCA the application shall call i2c\_ioctl with command paramater DEV\_IOCTL\_I2C\_ENABLE\_GENERAL\_CALL\_ADDRESS. An application might use GCA to discover particular devices on the I2C bus. For example, an application may respond to GCA with its address (and possibly its type).

Buffered messages do not capture the originating remote master transmitter address. So, while each message is loaded into and self-contained within an individual buffer, the sum of messages from all remote master transmitters will share the available buffer array. The number of buffers to allocate in the array is given by devConfig configDRV\_I2C\_BUFFER\_NUM; and the size of each individual buffer by configDRV\_I2C\_BUFFER\_SZ.

###### Slave Transmit role handling

Calling i2c\_open with parameter DEV\_MODE\_I2C\_MASTER\_SLAVE will access DEV I2C with both Master and Slave roles.

Slave Transmit role is the i2c activity of sending data to a remote Master receiver on-demand. DEV I2C shall either send an automatic 'who am i' reply consisting of device address (set previously through i2c\_ioctl) and type ('Agon'), if i2c\_open is not given an application callback parameter); or given a callback parameter, here named i2cHandler, DEV I2C will invoke it instead.

i2cHandler shall run in the context of a top-priority ISR Handler Task (see interrupt handling below). This shall execute while the I2C-bus is idled (by clock extending), so i2cHandler SHALL be prompt, in order to minimise the time it occupies the bus. i2cHandler SHALL respond to the remote Master by calling i2c\_writes. Data sent via i2c\_writes can be application-specific.

Slave Transmit identifies neither the remote master address nor its type. Any handshake or protocol needs to be established by application-specific design.

DEV I2C shall not provide buffers for Slave Transmit, as the task-provided parameters to i2c\_writes are sufficient, and avoid the need for memcpy.

###### Interrupt handling

Design of the ISR Handler Task would be based on that initially foreseen for DEV UART. This would allow interrupt handling to be passed to a highest priority task, and would enable concurrent interrupts from other devices.

BUT as with the UART, the eZ80 I2C interrupt hardware also needs the event cleared down before exiting the ISR, otherwise an infinite trigger results. Such that the entire handling must be run within the ISR context.

###### I2C Locking Bug

Refer to Zilog UP0049 "Errata for eZ80F92" Issue #4. A pulse on the SCL line prior to a START condition or after a STOP condition causes the I2C bus to lock. This situation occurs regardless of the I2C control register ENAB settings (I2C\_CTL). If this situation occurs, an I2C software reset does not unlock the I2C bus.

Workaround: To prevent a lock from occurring: it is possible to completely disable the I2C block prior to any bus activity using the Clock Peripheral Power-Down Register 1 (CLK\_PPD1). Disable the I2C block before setting ENAB in the I2C Control register.

##### DEV SPI Description

The eZ80 SPI device has only a single-place Transmit Holding Register (THR); there is no FIFO. Unlike the UART, only the eZ80 can bus-master the SPI device for any and all transmit and receive operations.

If we were to use interrupts, we would spend more time context switching than performing useful work. For this reason, there is no point supporting interrupts, rather we byte bash. The application can decide how many bytes to transmit / receive per DEV SPI call.

##### DEV UART Description

DEV UART provides a simple (open, close, read, write) API to access the Agon UART1 device. It allows Agon to connect with other end-points or devices, using the RS-232 standard with TTL (more specifically 3.3v) voltage levels.

###### Modes

Key to operation of UART1 is the mode argument passed to uart\_open.

DEV UART provides the ability for a FreeRTOS application to open a UART1 connection in one of seven different modes. These modes require allocation of different pins, and support different data flow control methods. Two modes support long-distance connections through external DCE devices; data rates decrease with distance due to signalling capacitance and noise. One mode supports end-point addressing using a DCE; all other modes support address-less point-to-point connection.

Once uart\_open is called (and uart\_ioctl for the one addressed end-point mode), DEV UART provides all the required operation for uart\_reads and uart\_writes. Only the one addressing mode requires application task uart\_ioctl programming in connection establishment prior to reading and writing.

DEV UART manages all other modes automatically, using the simple uart api to send and receive data.

|  |  |
| --- | --- |
| Mode | Summary |
| DEV\_MODE\_UART\_NO\_FLOWCONTROL | DTE<->DTE: rx/tx crossover wiring; no flow control |
| DEV\_MODE\_UART\_SW\_FLOWCONTROL | DTE<->DTE: rx/tx cross-over wiring; Xon / Xoff software flow control |
| DEV\_MODE\_UART\_HALF\_NULL\_MODEM | DTE<->DTE: rx/tx crossover, RTS/CTS crossover wiring; RTS/CTS hardware flow control |
| DEV\_MODE\_UART\_FULL\_NULL\_MODEM | DTE<->DTE: rx/tx crossover, RTS/CTS crossover, DTR/DSR+DCD crossover wiring; RTS/CTS hardware flow control |
| DEV\_MODE\_UART\_HALF\_MODEM | DTE<->DCE: rx, tx, rts, cts straight-through wiring; RTS/CTS hardware flow control |
| DEV\_MODE\_UART\_FULL\_MODEM | DTE<->DCE: rx, tx, RTS, CTS, DTR, DSR, DCD, RI straight-through wiring; RTS/CTS hardware flow control. End-point addressing. |
| DEV\_MODE\_UART\_LOOPBACK | DTE loopback, no wiring |

All seven modes enable full duplex (simultaneous tx and rx data transception) between end-points.

Four modes support point-to-point connected peer DTEs (data terminal equipment, such as Agon to a laptop) over short distances. These four modes differ in wiring and flow control methods. All four support point-to-point connection only – no end-point addressing.

DEV\_MODE\_UART\_NO\_FLOWCONTROL

This is comparable to the mode that UART0 is operated in, between the eZ80 CPU and the ESP32 VDP.

Pins:

* Requires just pins 17 (Tx) & 18 (Rx), and does without handshaking.

Use Case:

* 'No Modem' direct DTE<->DTE data transfer; low capacitance (short wires)

Wiring:

* DTE1.Tx --> DTE2.Rx cross-over
* DTE1.Rx <-- DTE2.Tx

Protocol:

* There is no DTE-DTE set-up; rather DTEs are 'always on'.
* Transmission and Reception are not controlled by hardware status, and are simultaneously possible.
* There are no end-to-end flow control command bytes. This incurs zero overhead, but the receive buffer may overflow and lose data.
* UART1\_MCTL is fixed at 0x00, with /RTS and /DTR de-asserted (high)
* There should be no UART\_IIR\_MODEMSTAT events.

DEV\_MODE\_UART\_SW\_FLOWCONTROL

Uses software flow control with Xon / Xoff handshaking. This operates by embedding ASCII characters DC1 and DC3 respectively directly into the data stream.

Pins:

* Also requires pins 17 and 18 only.

Use Case:

* 'No Modem' direct DTE<->DTE data transfer; low capacitance (short wires)

Wiring:

* DTE1.Tx --> DTE2.Rx cross-over
* DTE1.Rx <-- DTE2.Tx

Protocol:

* There is no DTE-DTE set-up; rather DTEs are 'always on'.
* Transmission and Reception are simultaneously possible.
* Xon / Xoff flow control command bytes can be sent in-stream from either end to the other. These are used to put the sender on pause to avoid buffer overflow. This incurs overhead as the receiver has to parse every byte received, but is less likely to result in data loss.
* UART1\_MCTL is fixed at 0x00, with /RTS and /DTR de-asserted (high)
* There should be no UART\_IIR\_MODEMSTAT events.

DEV\_MODE\_UART\_HALF\_NULL\_MODEM

Uses hardware flow control with RTS/CTS handshaking.

Pins:

* Requires pins 19 (RTS) and 20 (CTS) for hardware handshaking in addition to pins 17 (tx) and 18 (rx).

Use Case:

* 'No Modem' direct DTE<->DTE signalling; low capacitance (short wires)

Wiring:

* DTE1.Tx --> DTE2.Rx cross-over
* DTE1.Rx <-- DTE2.Tx
* DTE1.RTS --> DTE2.CTS cross-over
* DTE1.CTS <-- DTE2.RTS

Protocol:

* There is no DTE-DTE set-up; rather DTEs are 'always on'.
* Reception is always possible; transmission is by /RTS-/CTS handshake.
* With DEV\_MODE\_UART\_HALF\_MODEM, hardware takes the place of in-stream Xon / Xoff software command bytes. This exchanges the overhead of in-stream data byte parsing for managing hardware signalling. The liklihood of data loss is comparable to that of Xon / Xoff.
* With crossover wiring /RTS shall be asserted (low) when the DTE is Ready to RECEIVE. Think of it as Remote To Send.
* /RTS (Remote To Send) shall be de-asserted (high) when the DTE is ready to TRANSMIT.
* Prior to transmission, /RTS must be de-asserted (high) and /CTS assertion tested (low).
* If during transmission /CTS becomes de-asserted (high), then transmission must pause, and LCTL.SB set to indicate a break.
* Once /CTS is re-asserted (low) SB shall be cleared and transmission resumed.
* On transmission completion or abandon, /RTS must be asserted (low), so that reception remains possible.

DEV\_MODE\_UART\_FULL\_NULL\_MODEM

Uses hardware flow control with RTS/CTS handshaking.

Pins:

* Requires pins 21 (DTR), 22 (DSR) and 23 (DCD) in addition to pins 17 (tx), 18 (rx), 19 (RTS) and 20 (CTS).

Use Case:

* 'No Modem' direct DTE<->DTE signalling; low capacitance (short wires)

Wiring:

* DTE1.Tx --> DTE2.Rx cross-over
* DTE1.Rx <-- DTE2.Tx
* DTE1.RTS --> DTE2.CTS cross-over
* DTE1.CTS <-- DTE2.RTS
* DTE1.DTR --> DTE2.DCD three-way cross-wire
* '----> DTE2.DSR
* DTE1.DSR <-- DTE2.DTR
* DTE1.DCD <----'

Protocol:

* The DTE end-points handshake connection set-up and take-down through the additional /DTR, /DSR and /DCD wires; 'switched on'.
* Once the connection is 'set-up', reception is always possible and transmission is by /RTS-/CTS handshake.
* /DTR shall be asserted (low) when the DEV UART is opened, and de-asserted (high) when DEV UART is closed
* With crossover wiring /RTS shall be asserted (low) when the DTE is Ready to RECEIVE. Think of it as Remote To Send.
* Prior to transmission, /RTS must be de-asserted (high) and /CTS assertion tested (low).
* If during transmission /CTS becomes de-asserted (high), then transmission must pause, and LCTL.SB set to indicate a break.
* Once /CTS is re-asserted (low) SB shall be cleared and transmission resumed.
* If during transmission either /DCD or /DSR become de-asserted (high), then transmission must be abandoned.
* On transmission completion or abandon, /RTS must be asserted (low)

Two modes enable connection over longer distances requiring external DCE (data communication equipment[[1]](#footnote-1)). DCE-type is not specified and can include various network signal carrier technologies (wire, wireless). Types include “Modem”, “Bridge”, “Switch” and “Router”.

One of these two Modem modes supports a point-to-point connection.

DEV\_MODE\_UART\_HALF\_MODEM

Uses hardware flow control with RTS/CTS handshaking.

Pins:

* Uses the same pins 17, 18, 19, 20 as HALF\_NULL\_MODEM, but wired straight-through.

Use Case:

\* Both DTE<->DCE and DTE<->DTE control flow signalling; long-distance wiring.

Wiring:

* DTE1.Tx --> DCE1.Tx straight-through
* DTE1.Rx <-- DCE1.Rx
* DTE1.RTS --> DCE1.RTS straight-through
* DTE1.CTS <-- DCE1.CTS

Protocol:

* There is no DTE-DTE set-up; rather DTEs are 'always on'. (DTE knows nothing about the connection state of the DCE 'modem', which may be 'always on' or 'woken up'.)
* Reception is always possible; transmission is by /RTS-/CTS handshake.
* In addition to DTE endpoint /RTS - /CTS control-flow management, the 'modems' or 'bridges' that sit in-between the DTE end-points may manage the control flow (by de-asserting /CTS) depending on signal quality, further reducing chance of data loss.
* /RTS (Request To Send) shall be asserted (low) when the DTE is readyto TRANSMIT.
* Prior to transmission, /RTS must be asserted (low) and /CTS assertion tested (low).
* If during transmission /CTS becomes de-asserted (high), then transmission must pause, and LCTL.SB set to indicate a break.
* Once /CTS is re-asserted (low) SB shall be cleared and transmission resumed.
* On transmission completion or abandon, /RTS must be de-asserted (high)

The other of these two Modem modes supports addressed end-point connection – the only mode to do so.

DEV\_MODE\_UART\_FULL\_MODEM

Uses hardware flow control with RTS/CTS handshaking. Is the only mode to support remote end-point addressing, immediately following the uart\_open call, achieved through the uart\_ioctl API and short control packets (like the Hayes “AT” command set) sent into the data stream.

Pins:

* Uses all the pins 17-23 like DEV\_MODE\_UART\_FULL\_NULL\_MODEM, and in addition pin 24 (RI).

Use Case:

* DTE<->DCE switching; and DTE<->DTE control flow signalling; long-distance wiring.

Wiring:

* DTE1.Tx --> DCE1.Tx straight-through
* DTE1.Rx <-- DCE1.Rx
* DTE1.RTS --> DCE1.RTS straight-through
* DTE1.CTS <-- DCE1.CTS
* DTE1.DTR --> DCE1.DTR straight-through
* DTE1.DSR <-- DCE1.DSR
* DTE1.DCD <-- DCE1.DCD
* DTE1.RI <-- DCE1.RI

Protocol:

* The DTE handshakes connection set-up and take-down with its local DCE 'modem' through the additional /DTR, /DSR, /DCD and /RI wires.
* This differs to the fixed point-to-point connection models of other modes by allowing switched networks with multiple end-points.
* /DTR shall be asserted (low) when the DTE wishes to write or when an incoming /RI is indicated; either of which initiate 'call set-up'.
* /DTR shall be de-asserted on transmission end, or when one of /DSR or /DCD is de-asserted which initiate 'call take-down'.
* Once a connection is set-up, reception is always possible; transmission is by RTS/CTS handshake.
* Reception is possible with /RTS asserted (low) or de-asserted (high)
* /RTS (Request To Send) shall be asserted (low) when the DTE is ready to TRANSMIT.
* Prior to transmission, /RTS must be asserted (low) and /CTS assertion tested (low).
* If during transmission /CTS becomes de-asserted (high), then transmission must pause, and LCTL.SB set to indicate a break.
* Once /CTS is re-asserted (low) SB shall be cleared and transmission resumed.
* If during transmission either /DCD or /DSR become de-asserted (high), then transmission must be abandoned.
* On transmission completion or abandon, /RTS must be de-asserted (high).

One mode supports wireless connection, by programming the eZ80 UART in loopback mode. This mode is useful for testing on the Agon alone, with no peer end-point.

DEV\_MODE\_UART\_LOOPBACK

In this mode the eZ80 UART is programmed in loopback mode. Uses hardware flow control with RTS/CTS handshaking. Follows DEV\_MODE\_UART\_FULL\_MODEM without addressing.

Pins:

* Requires no pin wiring. In loopback mode all the pins 17-24 are logically assigned like DEV\_MODE\_UART\_FULL\_MODEM.

Use Case:

* Loopback testing
* Operating using Agon standalone
* As a form of buffer or message harness between tasks

Wiring:

* None (UART programmed as a loopback):
  + DTE1.Tx --> DTE1.Rx
  + DTE1.RTS --> DTE1.RTS
  + DTE1.DTR --> DTE1.DSR
  + DTE1.OUT2 --> DTE1.DCD
  + DTE1.OUT1 --> DTE1.RI

Protocol:

* Follows DEV\_MODE\_UART\_FULL\_MODEM without addressing

###### Buffered and Unbuffered methods

Bit or’ing either DEV\_MODE\_UNBUFFERED (the default) or DEV\_MODE\_BUFFERED with one of the above modes as an argument to uart\_open, DEV API shall allow either buffered or non-buffered methods for sending and receiving.

With the non-buffered method, a calling task will be blocked until either transmission is complete or an error occurs. This method is best used with a task dedicated to uart i/o. Whereas with the buffered method, the calling task will not be blocked, but continue to run concurrently with uart transmission. This method is best used with a task design that loops around a number of independent activities. Either method can be used with two separate FreeRTOS tasks, one for sending and another for receiving.

###### Connection establishment

DEV\_MODE\_UART\_FULL\_MODEM is the only mode that will engage in a connection establishment protocol. Each 'Modem' defines its own connection setup language and protocol (such as Hayes "AT" commands, or TCP/IP packets), so it is not practical to embed in DEV UART.

Rather, DEV UART supports connection setup as follows. To initiate a connection:

1. uart\_open in DEV\_MODE\_UART\_FULL\_MODEM (or DEV\_MODE\_UART\_LOOPBACK)
2. negotiate the call set-up
3. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=1
4. uart\_ioctl DEV\_IOCTL\_UART\_SET\_DTR param=1
5. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DSR until param = 1
6. uart\_write "connect request data" (iterate)
7. uart\_read "modem confirmation data" (iterate)
8. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DCD until param = 1
9. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=0
10. uart\_read and uart\_write the remote end-point
11. [optional] negotiate the call tear-down:
    1. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=1
    2. uart\_write "disconnect request data" (iterate)
    3. uart\_read "modem confirmation" (iterate)
    4. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=0
12. uart\_close, which performs the following:
    1. uart\_ioctl DEV\_IOCTL\_UART\_SET\_DTR param=0
    2. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DCD until param = 0
    3. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DSR until param = 0

To respond to an incoming call, differs in step 2:

1. negotiate the call parameters
   1. uart\_poll to test modem\_status.RI

or uart\_ioctl DEV\_IOCTL\_UART\_GET\_RI if param = 1

* 1. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=1
  2. uart\_ioctl DEV\_IOCTL\_UART\_SET\_DTR param=1
  3. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DSR until param = 1
  4. uart\_read "modem indication data" (iterate)
  5. uart\_write "connect response data" (iterate)
  6. uart\_ioctl DEV\_IOCTL\_UART\_GET\_DCD until param = 1
  7. uart\_ioctl DEV\_IOCTL\_UART\_WRITE\_MODEM param=0

## References

[Agon] <https://github.com/TheByteAttic/AgonLight> Agon Light hardware

[AgonCluster] <https://github.com/julian-rose/FreeRTOS-for-Agon/blob/main/FreeRTOSv202212.01-LTS/FreeRTOS/Docs/DEV%20API%20-%20Extension%20Interface%20and%20MOS%20analysis.docx> section outlining Agon Cluster Networks

[FreeRTOS Kernel] <https://www.freertos.org/RTOS.html> About the FreeRTOS kernel

[MOS] <https://agonconsole8.github.io/agon-docs/MOS/> Machine Operating System for Agon Light and compatibles.

[MOS API] <https://agonconsole8.github.io/agon-docs/MOS-API/#the-mos-api>

[RS-232] <https://en.wikipedia.org/wiki/RS-232>

## Other References

[Hayes] <https://en.wikipedia.org/wiki/Hayes_AT_command_set>

[tcpser] <https://github.com/go4retro/tcpser> Hayes Modem emulation software for PC. Thought about using this as a gateway. Would still need a remote end-point to connect with.

1. 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-1)