# 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\_read, i2c\_write |
|  | 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 |
|  | configUSE\_DEV\_SAFEGUARDS  configUSE\_FAST\_INTERRUPTS |
|  | configDRV\_UART\_BUFFER\_SZ  configDRV\_UART\_UNBUFFERED\_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 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

[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)