

# P2 EXTERNAL MEMORY DRIVER DOCUMENTATION (Release 0.9b)

## Introduction

This driver suite provides external memory capabilities for the Propeller 2 supporting a selection of RAM and flash memory devices. It enables multiple P2 COGs to access different memory devices on both shared and separate memory buses. It includes a SPIN2 based API to initialize and map the memory devices into a common address space beyond the HUB memory space. It also includes dynamic configuration settings to ensure Quality of Service (QoS) when accessing the memory devices. PASM2 client COGs can also control this driver directly, in cases where SPIN2 code is not needed. HUB RAM based mailboxes are used to issue the external memory requests, and are separated by both COG and bus so that multiple independent requests can be created simultaneously by multiple COGs. COG requests are serviced according to a nominated priority, or by a round-robin scheduler in the driver. Larger burst transfers are also fragmented to remain within the maximum chip select time for the given memory type to avoid refresh problems.

This driver suite and its control API offers an extensive list of features including this following:

- manages up to 16 different sized RAM and/or Flash devices per shared data bus
- each bus provides up to 256MB of addressable memory
- requests are serviced by shared mailbox polling, with round-robin and strict priority algorithms
- multiple independent memory buses are supported (limited by pin count or available COGs)
- For HyperRAM/HyperFlash devices, the device registers are also accessible
- transfer burst sizes can be strictly controlled, and bursts are fragmented automatically
- optional notification via COGATN at end of request being serviced with the mailbox update
- error codes are reported for all failed requests
- linked lists of multiple requests can be issued in one operation to reduce overhead (effectively having the driver behave like a simple DMA engine for automating multiple memory transfers)
- bytes, words and long transfers to/from HUB memory and external memory are all supported
- arbitrary length read/write burst/fill transfers supported (byte granular) for highest performance
- memory copy operations are supported between the same or different devices on the bus
- graphics based fill and copy operations are supported with adjustable image width and height
- exclusive access to HyperFlash is provided during erase/programming operations by any COG
- Hyper devices transfer bytes at P2 sysclk/2 rates for reliable reads and writes, and optionally at sysclk/1 rates for maximum performance (sysclk/1 writes may require extra HW support)
- includes a simple initialization method for the Parallax HyperRAM/HyperFlash breakout module or PSRAM P2 Edge, and a general purpose mapping API exists for custom implementations
- selectable control pins for the memory devices, with optional reset strobe support upon startup
- adjustable per device latency and read timing delay profile for optimizing transfer performance
- contains HyperFlash erase/programming routines in the SPIN2 API
- compatible with both FlexSpin and Parallax SPIN2 tools

# Memory Driver API Summary

This section lists the APIs available in the memory driver for the different functions. Only the method names are shown here, so refer to the code for all the arguments and further details.

## Memory initialization

Any PASM2 bus driver that accesses its external memory bus first needs to be initialized once with a valid memory map. This can be done automatically, by calling one of these convenient setup API's for the different type(s) of memory in use:

- ***initHyperDriver*** - single call to map the Parallax P2-EVAL HyperRAM/HyperFlash breakout board's device(s) into the external memory address space and to start its bus driver
- ***initP2EdgeDriver*** - single call to map the Parallax P2 Edge with 32MB of PSRAM into the external memory address space and to start its PSRAM driver (for a 16 bit wide configuration)
- ***initSramDriver*** - single call to map an 8 bit static RAM device into the external memory address space and to start its bus driver

Or, if you want finer control of the device pins, mapping and the number of device banks, you can call any of these mapping methods at least once and then also start the bus driver manually:

- ***mapHyperRAM*** - maps a HyperRAM device to a bus and address
- ***mapHyperFlash*** - maps a HyperFlash device to a bus and address
- ***mapPsrAm*** - maps a PSRAM device to a bus and address
- ***mapSram*** - maps a static RAM device to a bus and address
- ***mapHubRam*** - maps HUB RAM to a bus and address (intended only for driver testing/debug)

The following call should be made once per bus after mapping all its banked devices with the APIs above:

- ***start*** - start the PASM2 driver for a memory bus containing mapped devices using the same type of memory (HyperMem, PSRAM, SRAM, HUB etc). Optional startup flags can be passed here.

Either of the two different initialization sequences above will spawn a PASM2 bus driver COG to manage all the requests to the device(s) on that data bus. Devices mapped on different data buses will spawn additional bus driver COG(s) in those situations where there are multiple independent memory buses attached to the P2. The total number of buses supported is controlled by setting the ***MAX\_INSTANCES*** constant in the SPIN2 memory driver. Its default value (1) can be changed to match the system's requirements. Increasing it will increase the driver state memory required so only set it to the maximum number of buses you will be using.

## Configuring COG polling & QoS parameters

Once the PASM2 driver is started, by default all other COGs will share external memory access opportunities using round robin mailbox polling in the driver. However it is also possible to dynamically reconfigure the number of COGs that can access the memory devices supported by the driver and the manner in which their requests are serviced, to better match the application's requirements.

Removing any extra COGs that do not require access to external memory helps to eliminate excess mailbox polling and reduces the service latency. Removal of unwanted COGs can be done once all the particular COG IDs that will need to access this memory become known following system startup of other COGs etc.

To remove a group of COGs from being serviced by the driver, use this API below. Note that for these methods the COGs are identified by an 8 bit mask, and not directly by a COG ID.

- ***removeCogs*** - removes a set of COG mailboxes from those being polled by the driver for a given bus

To later include COG mailboxes for polling and to specify their QoS servicing attributes, use this:

- ***setupQoS*** - sets up QoS parameters and other flags for the selected COG(s)

This allows the COG's maximum transfer burst sizes and their mailbox polling priorities to be specified, and how they are to be managed when servicing their requests. See the later section covering the COG QoS Parameter Structure for further details on how to set things up.

Several flags can be passed to this API which affect how the nominated COG(s) are processed:

- ***F\_ATN*** enables additional notification with COGATN on completion of each request
- ***F\_PRIORITY*** enables priority polling, otherwise round-robin polling is used for these COGs
- ***F\_LOCKED*** completes full burst transfers from a COG before any other COGs get serviced
- ***F\_STALL*** stalls round robin COGs if accessing flash that is exclusively locked by another COG

A priority value from 0-7 is provided when the ***F\_PRIORITY*** polling is active, along with the transfer burst size used before any fragmentation happens that would re-trigger further polling. COGs with higher priorities are serviced first, and any COGs with equal priorities are serviced in COG ID order. Lower priority or round-robin polled COGs will ONLY begin to be serviced by the poller when all higher priority COGs are not currently requesting or being serviced. This priority feature along with setting suitable burst sizes allows guaranteed real-time support for video/audio clients in the presence of lower priority requests.

The minimum applicable burst size must be at least 4 bytes to enable a long sized data transfer. Burst sizes below this are not supported and will cause the COG to be removed from service.

## Driver shutdown

To shutdown a memory bus after it has been created and for freeing its resources, the following method should be used. This stops and releases the driver COG for a given memory bus. Any pending requests can either be terminated immediately or optionally waited on until completion.

- ***shutdown*** - shuts down a memory bus, optionally waiting for completion.

## Miscellaneous utility methods

Several other helper or utility methods are provided to assist with various client operations.

- ***getMailboxAddr*** - find HUB start address of the 3 long mailbox for a given bus & client COG ID
- ***getDriverCogID*** - returns the driver COG ID that is managing a given bus
- ***getDriverLockID*** - returns the ID of the lock that the driver is using
- ***getResult*** - check/wait for mailbox success/error result
- ***getFlashLocked*** - returns whether flash is locked and which COG locked it for exclusive use
- ***getMaxBurst*** - utility to help compute a maximum burst size for a given duration
- ***setDelayFrequency*** - adjusts input delay for P2 operating frequency, if it changes after startup

## Driver flags

Some drivers interpret special startup flags that can customize the driver further and are used for experimenting or extending the features in the driver. These flags currently include the following masks:

**F\_FASTREAD** - enables faster sysclk/1 operation for Hyper memory reads  
**F\_FASTWRITE** - enables faster sysclk/1 operation for Hyper memory writes  
**F\_UNREGCLK** - enables unregistered clock pin timing  
**F\_EXPANSION** - enables (experimental) graphics acceleration options in driver

## Reads

- ***readByte*** - reads and returns a single byte from external memory or any negative error code
- ***readWord*** - reads and returns a single word from external memory or any negative error code
- ***readLong*** - reads and returns a single long from external memory
- ***read*** - reads a burst of external memory into HUB memory
- ***readReg*** - reads a register from HyperRAM or HyperFlash
- ***readByteErr*** - reads a byte and returns the result and any error separately as a result pair
- ***readWordErr*** - reads a word and returns the result and any error separately as a result pair
- ***readLongErr*** - reads a long and returns the result and any error separately as a result pair

## Writes

- ***writeByte*** - writes a single byte to external RAM
- ***writeWord*** - writes a single word to external RAM and HyperFlash
- ***writeLong*** - writes a single long to external RAM
- ***write*** - writes a burst of HUB RAM data to external memory
- ***writeReg*** - writes a HyperRAM or HyperFlash register

## Read-modify-writes

- ***readModifyByte*** - reads and alters a byte of memory in external RAM using an 8 bit mask
- ***readModifyWord*** - reads and alters a word of memory in external RAM using a 16 bit mask
- ***readModifyLong*** - reads and alters a long of memory in external RAM using a 32 bit mask

## Fills & Copy

Ranges of external memory can be filled with data of the given size by using the following:

- ***fill*** - a generalized fill that fills bytes, words or longs based on a data size argument  
or
- ***fillBytes*** - fills a block of bytes in external RAM with a nominated byte value
- ***fillWords*** - fills a block of words in external RAM with a nominated word value
- ***fillLongs*** - fills a block of longs in external RAM with a nominated long value

External memory blocks can be copied between or within memory devices on the same bus (in increasing address order only), or between devices on different buses, with read/write bursts passing through an intermediate hub buffer in both cases. HyperFlash cannot be directly copied into, only RAM devices can be the destination of the copy.

- ***copyBus*** - copies bytes between/within memory banks on the same bus
- ***copy*** - copies bytes from external memory to external RAM on any bus

## Graphics Operations

There are several methods that are specifically designed to transfer image data in external memory and these leverage capabilities built into the PASM2 driver that allow rectangular regions with given width, height and scan line spacing to be transferred in a single mailbox operation. The graphics fill operations can also be used to support drawing vertical or horizontal lines.

- ***gfxCopyImage*** - copies a graphics image between or within external RAM (automatically via a HUB buffer)
- ***gfxReadImage*** - reads a graphics image from external memory into HUB RAM
- ***gfxWriteImage*** - writes a graphics image from HUB RAM into external RAM
- ***gfxFill*** - a generalized graphics fill that fills bytes, words or longs depending on a size argument or
- ***gfxFillBytes*** - fills a rectangular image area in external RAM with a nominated byte value
- ***gfxFillWords*** - fills a rectangular image area in external RAM with a nominated word value
- ***gfxFillLongs*** - fills a rectangular image area in external RAM with a nominated long value

## Request Lists

Request lists containing request items can be optionally prepared using ***copyBus*** or any of the fill or graphics related methods mentioned above whenever a non-zero list pointer argument is supplied during the call. Burst reads and writes are also supported in lists by preparing their requests list items using either of these methods, instead of the usual read/write APIs identified earlier:

- ***writeList*** - sets up a burst write list item for running from a request list
- ***readList*** - sets up a burst read list item for running from a request list

Multiple request list items can be linked together to build the full request list for a COG. The methods that setup list items also return the HUB address of the link field within the newly created list item, which is useful for chaining them together.

The actual execution of a request list can either be issued in a non-blocking / asynchronous manner where the requesting COG can continue operating while its request list is being serviced in the background, or it can be issued synchronously and block until the entire request is complete. Non-blocking operation can work well with COGATN notification. Use the following API to execute a list.

- ***execList*** - executes an already prepared request list, with a blocking/non-blocking option

## HyperFlash specific

HyperFlash has its own set of operations and complex sequences involving erasure and re-programming. The methods below support these actions but can still be extended by the client if needed for accessing the advanced HyperFlash features like enabling per sector protection, etc.

- ***readFlashInfo*** - reads HyperFlash device information
- ***readFlashStatus*** - reads the HyperFlash status register
- ***clearFlashStatus*** - clears the HyperFlash status register
- ***eraseFlash*** - erases either a single sector or the entire HyperFlash memory
- ***pollEraseStatus*** - checks on the HyperFlash erase status during the non-blocking erase mode
- ***programFlash*** - writes a block of HUB RAM into HyperFlash memory
- ***programFlashByte*** - writes a single byte into HyperFlash memory
- ***programFlashWord*** - writes a single word into HyperFlash memory
- ***programFlashLong*** - writes a single long into HyperFlash memory
- ***lockFlashAccess*** - locks the HyperFlash device for exclusive use by a single COG
- ***unlockFlashAccess*** - unlocks the HyperFlash device for normal use by other COGs
- ***readFlashICR*** - reads Interrupt Configuration Register in HyperFlash
- ***readFlashISR*** - reads Interrupt Status Register in HyperFlash
- ***readFlashNVCR*** - reads Non-Volatile Configuration Register in HyperFlash
- ***readFlashVCR*** - reads Volatile Configuration Register in HyperFlash
- ***writeFlashICR*** - writes Interrupt Configuration Register in HyperFlash
- ***writeFlashISR*** - writes Interrupt Status Register in HyperFlash
- ***writeFlashVCR*** - writes Volatile Configuration Register in HyperFlash

When HyperFlash is not being programmed, erased or having its registers accessed, then all COGs can read it. However if it is being accessed using a complex register control sequence, a lock is required to prevent other COGs from affecting things part way through the sequence. Both ***lockFlashAccess*** and ***unlockFlashAccess*** are used to gain or free exclusive flash access by a single COG. This memory driver API will call these automatically as needed during register access, erasure and reprogramming. Other COGs that attempt to access the flash in an exclusive state will either block, or return a flash busy error until the flash exclusion lock is released.

Erasing HyperFlash is slow so it can be run in either a blocking mode or a non-blocking mode. If non-blocking erase is requested then ***pollEraseStatus*** MUST continue to be called in order to identify when the erase operation has completed and can be returned into its usual standby ready state, and no further HyperFlash request activity can be made until then by the COG.

## Advanced Configuration

The methods below are provided only for advanced driver configuration, and would not typically be required for general use while the driver is operating if the default settings are used. These can be used to experiment with timing and for manually tuning performance by users with more detailed knowledge of their systems and external memory device behaviour. It can also be used to access additional internal driver state if reading back the configuration, or for debug purposes.

Delays can be configured per device, and bursts and latencies can be adjusted for tuning.

- ***setBurst*** - set a memory device's maximum transfer burst size before fragmentation occurs
- ***getBurst*** - get transfer burst size assigned to a memory device
- ***setDelay*** - set a memory device's read delay
- ***getDelay*** - get a memory device's configured read delay
- ***getFlags*** - get a memory device's configured flag state
- ***getQoS*** - get the QoS settings for a given COG on a given memory bus
- ***getSize*** - get the size of a mapped external memory device in bytes
- ***getBusCount*** - returns the total number of external memory buses supported by the driver
- ***getBus*** - obtains the bus used by an address if it is mapped
- ***getPinParameters*** - returns the provisioned Pin Parameter Long for a bank on a bus
- ***getBankParameters*** - returns the provisioned Bank Parameter Long for a bank on a bus
- ***getFlashLockedCog*** - returns which COG has locked the HyperFlash device on a bus
- ***setDelayProfile*** - configures a custom delay profile to be applied if the frequency changes
- ***getDefaultProfile*** - retrieves a default delay profile for Flash or RAM
- ***readRamIR*** - reads an Identification Register in HyperRAM
- ***readRamCR*** - reads a Control Register in HyperRAM
- ***writeRamCR*** - writes a Control Register in HyperRAM
- ***setRamLatency*** - sets up both driver and device latency clocks for HyperRAM
- ***setFlashLatency*** - sets up both driver and device latency clocks for HyperFlash
- ***getDriverLatency*** - get the driver's latency for a device
- ***dump*** - dumps the PASM2 based driver's COG+LUT RAM state to HUB RAM, for debugging



## PASM2 Driver Mailbox Details

Mailbox based transactions are automatically controlled internally by the SPIN2 API, however for PASM2 clients that wish to access the external memory directly, their memory requests will need to be setup manually in the mailbox allocated to the requesting COG, or in the mailbox allocated to the driver COG if they intend to issue control requests. This section details the mailbox.

### Mailbox Format and Usage

The mailbox for each COG is a 3 long structure stored in HUB RAM and is both written and read by the requesting client COG as well as by the driver COG. Its layout is shown below.

MAILBOX	PURPOSE
LONG 1	Triggers the request and provides an external memory address or other data
LONG 2	Contains write data or a HUB address, and the requested data is returned here
LONG 3	Optional mask or transfer count parameter depending on type of request

For the standard external memory data read or write requests the first mailbox entry (*LONG 1*) is divided up into the following fields:

Bit 31	Bits 30-28	Bits 27-24	Bits 23-0
A	Request Type	Bank / AddressHigh	External Memory Address Low

Bit31 (A) is set 1 to activate a new request by a client COG, and will be cleared by the driver COG when the request is complete and the mailbox is inactive. The requesting COG can choose to poll this mailbox address to determine whenever the last transaction has completed and to determine that the mailbox is now idle. New requests should not be issued while this A bit is set.

The request type identifies what action the driver will perform - it is described in the next section.

The Bank/Address High value is a 4 bit field that identifies which particular memory device on the bus will be accessed. For memory devices 16MB or smaller, its value uniquely identifies the device within a given bank, while for memory devices greater than 16MB in size some bank bits are treated as the higher bits of the external memory address and the device will then need to span multiple banks.

The External Memory Address (augmented with Address High bits if the device is sized > 16MB) is the actual external physical address which will be accessed within the external memory device.

The second mailbox entry (*LONG 2*) nominates the HUB address to be used as part of the burst transfer, or the data value to write to HUB (for both Writes and Read-Modify-Write operations).

The third mailbox entry (*LONG 3*) is used as a count of the number of separate byte/word/long writes (fill operations), or the number of bytes to transfer for burst reads/writes. For read-modify-write operations, it is instead used as a 32 bit mask applied to the data before being written back. If this mask value is 0, nothing will be written back to external memory (i.e. standard read), and if the mask is all ones (\$FFFFFFFF) then all relevant bits in the data field in the second long will replace the original value. In both cases the original value from external memory will be returned to the client COG. This is an atomic operation that cannot be pre-empted and as such can be useful for mutexes. Any zero bits in the mask will leave the corresponding data bit in external memory intact. Read-modify-write operations only apply to external RAM, not to Flash.

Once the request has been serviced by the driver, it will clear bit31 of mailbox *LONG 1* to indicate service completion. If the client COG has also been configured in the driver to receive ATN notifications on service completion, that will also occur at this time. If the request succeeded then the entire mailbox *LONG 1* will be cleared to zero and any data result is returned in mailbox *LONG 2*. If the request failed, the applicable (positive valued) error code will be written to mailbox *LONG 1*. Mailbox *LONG 3* is always left intact by the driver which can be useful for repeat reads.

Each COG has its own dedicated group of 3 mailbox longs for each memory bus, while each driver COG's mailbox is shared by all COGs and is used for their control requests. Exclusion needs to be enforced to prevent COGs from corrupting the control mailbox if it is already busy. The SPIN2 API already handles this with a P2 HUB lock protecting the control mailboxes, but any PASM2 clients also issuing control requests will need to manage this on their own, by sharing the same lock or by otherwise ensuring only one COG ever performs control requests at any time.

## Memory Requests

The request type is a 3 bit value that identifies the type of memory access to be performed by the driver. The mapping of these request type bits to the request action is shown below:

REQUEST	ACTION
%000	Read a byte from external memory, optionally update it with a masked value
%001	Read a word from external memory, optionally update it with a masked value
%010	Read a long from external memory, optionally update it with a masked value
%011	Read a block of bytes from external memory into HUB memory
%100	Write a byte to external memory, or fill a range with a byte value
%101	Write a word to external memory, or fill a range with a word value
%110	Write a long to external memory, or fill a range with a long value
%111	Write a block of bytes into external memory from HUB memory

Requests to read bytes, words or longs all use the following general mailbox format. The only difference is the size of the data returned in *LONG 2* and Write Value used for Read-Modify-Write as well as the request type value in the first long. This particular example below shows a word read, and only the least significant word of *LONG 2* and *LONG 3* would be used if Read-Modify-Write is applied. For normal reads, keep *LONG 3* set to zero and the write back will not occur.

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-0
LONG 1	1	%001	Bank	External Memory Source Address
LONG 2	Optional Write Value for RMW			
LONG 3	RMW Mask			

Requests to burst read from external memory to a given HUB address use the following format:

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-0
LONG 1	1	%011	Bank	External Memory Source Address
LONG 2	Hub Address			
LONG 3	Byte Count			

Requests to write or fill bytes, words or longs to an external memory destination address with a given data value use the following format. The Fill Count would be set to 1 for a single write. This particular example shows a long fill, the other fill sizes are similar but use different request bits.

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-0
LONG 1	1	%110	Bank	External Memory Destination Address
LONG 2	Data Value			
LONG 3	Fill Count			

Requests to burst write data bytes into external memory Destination Address starting from a given HUB Address use the following format:

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-0
LONG 1	1	%111	Bank	External Memory Destination Address
LONG 2	Hub Address			
LONG 3	Byte Count			

## Control Requests

The 3 bit request field mapping for the control mailbox transactions are shown below:

Mailbox *LONG 1* always has bits 3-0 set to the ID of the calling COG and includes an applicable bank spanning the device being accessed.

REQUEST	ACTION
%000	Gets current driver latency clocks for the given bank, puts 8 bit result into LONG 2
%001	Reads a 16 bit device register using LONGs 2 & 3, put result in LONG 2 (see below)
%010	Gets current burst, protection & delay parameters for the bank, result into LONG 2
%011	Copies current COG+LUT register memory into HUB RAM address in LONG 2
%100	Sets the driver latency clocks for a given bank using the 8 bit data in LONG 2
%101	Writes a 16 bit device register using address/data in LONGS 2 & 3 (see below)
%110	Set the burst size, flash protection & delay for a given bank using 32 bits in LONG 2
%111	Reloads & reconfigures all COG QoS parameters from existing HUB RAM structure

Reading the device registers uses this mailbox format. The RegisterAddr value holds the 16 bit register address which will be read from the device. The 16 bit result will be returned in *LONG 2*.

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-4	Bits 3-0
LONG 1	1	%001	Bank	Reserved (set to 0)	COG ID of requestor
LONG 2	\$E000 + RegisterAddr[31:19]				
LONG 3	(RegisterAddr[18:3] << 16) + RegisterAddr[2:0]				

Writing device registers uses this mailbox format. The RegisterAddr value is a 16 bit register address that will be written. The WriteDataValue is the data to be written to a device register.

MAILBOX	Bit 31	Bits 30-28	Bits 27-24	Bits 23-4	Bits 3-0
LONG 1	1	%101	Bank	Reserved (set to 0)	COG ID of requestor
LONG 2	(WriteDataValue << 16) + \$6000 + RegisterAddr[31:19]				
LONG 3	(RegisterAddr[18:3] << 16) + RegisterAddr[2:0]				

## Advanced Requests

More advanced requests such as graphics fills and copies are handled using request lists. A request list is a linked list that identify a series of request(s) the driver will process sequentially for the COG until either the list ends or an error occurs. At that time the driver then notifies the COG of its completion. As such it allows the client COG to prepare requests in advance that can be processed by the driver in the background without further involvement by the client COG. This frees the COG for continuing other work in parallel and improves performance.

To start a list the HUB address of the first request item in the request list is written into mailbox *LONG 2* and \$FFFFFFFF is written to mailbox *LONG 1* in the client COGs mailbox. This special all ones pattern in mailbox *LONG 1* identifies a request list is to be processed instead of otherwise starting a regular burst write to bank 15 at device address \$FFFFFFF. Doing that particular burst write is therefore excluded by this scheme but that specific restriction will not be particularly limiting or onerous. Note that writing this all ones pattern to the driver's control mailbox in *LONG 1* is not treated as a request list, but will continue to be handled as a regular control request (it will still reconfigure QoS parameters). No control requests can be put into request lists.

Once a request list triggered, the driver will begin processing the list, working through each request item in the list in order. The current list position will be written back to mailbox *LONG 2* as each request completes so progress can be monitored and it will be present if an error occurs.

The format of the request list items in HUB memory can use either a 4 long structure or an extended 8 long structure depending on the type of request.

The smaller 4 long request list item structure starts out with the same 3 long parameter sequence from the standard mailbox requests, followed by a fourth long. The MSB in the fourth long is used as a way to identify whether this request list item is using the 4 long or the 8 long structure, as well as a link to the next request item in the request list (or 0 when the list ends).

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/AddrHi (4 bits) + AddrLo (24 bits)
List Item+4		Hub Address/Data Value
List Item+8		Count
List Item+12	MSB = 0	Link to next request

If Bit31 of the fourth long is zero, it signifies that this request list item is using the shorter 4 long structure, and the remaining bits are the HUB address of the next request item in this list.

If Bit31 of the fourth long is one, it signifies this request uses the extended structure needed for more complex graphics and copy requests and the formats in those case are described in the different tables below.

## Linear Copy

For linear (non-graphics) copies between two external memory addresses, the request list item format is interpreted as below. Only external RAM can be the destination, while either external RAM or Flash devices can be the source.

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/SrcAddrHi (4 bits) + SrcAddrLo (24 bits)
List Item+4		Hub Address
List Item+8		Hub Buffer Size*
List Item+12	MSB = 1	Req (3 bits) + Bank/DstAddrHi (4 bits) + DstAddrLo (24 bits)
List Item+16		Total Bytes*
List Item+20		0
List Item+24		0
List Item+28		Link to next request

List Item + 0 represents the external memory source address being read from, and uses a read burst request with MSB =1

List Item + 4 represents the HUB address of an intermediate HUB transfer buffer used during copy

List Item + 8 represents the number of bytes to copy to/from HUB in each transfer burst

List Item + 12 has MSB=1 and the external memory destination address for the copy operation, and uses a burst write request

List Item + 16 contains the total number of bytes to copy between external memory addresses

List Item + 20 contains zero

List Item + 24 contains zero

List Item + 28 contains a HUB address of the next request list item (or zero to end the list)

**Note \*:** Total Bytes and Hub Buffer Size should not be zero.

## Graphics Copy

For graphics image copies between two external memory addresses, the request list item format is this:

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/SrcAddrHi (4 bits) + SrcAddrLo (24 bits)
List Item+4		Hub Address
List Item+8		Hub Buffer Size*
List Item+12	MSB = 1	Req (3 bits) + Bank/DstAddrHi (4 bits) + DstAddrLo (24 bits)
List Item+16		Scan Lines*
List Item+20		Scan line offset for DstAddr
List Item+24		Scan line offset for SrcAddr
List Item+28		Link to next request

List Item + 0 represents the external memory source address being read from, and uses a read burst request with MSB = 1

List Item + 4 represents the HUB address of an intermediate HUB transfer buffer used during copy

List Item + 8 represents the number of bytes to copy per scan line (controls image width)

List Item + 12 has MSB=1 and the external memory destination address for the copy operation and includes a burst write request

List Item + 16 contains the number of scan lines to read (controls image height)

List Item + 20 contains an offset to add to the destination memory address per scan line copied

List Item + 24 contains an offset to add to the source memory address per scan line copied

List Item + 28 contains a HUB address of the next request list item (or zero to end the list)

**Note** \*: Scan lines and Hub Buffer Size should not be zero. HyperFlash can be the source but cannot be the destination.

## Graphics Read

For graphics image reads from external memory into HUB, the request list item format is this:

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/SrcAddrHi (4 bits) + SrcAddrLo (24 bits)
List Item+4		Hub Address
List Item+8		Hub Buffer Size*
List Item+12	MSB = 1	0
List Item+16		Scan Lines*
List Item+20		Scan line offset for Hub Address
List Item+24		Scan line offset for SrcAddr
List Item+28		Link to next request

List Item + 0 represents the external memory source address being read, and uses a read burst request with MSB = 1

List Item + 4 represents the HUB address to write the data

List Item + 8 represents the number of bytes to read per scan line (controls image width)

List Item + 12 is zero apart from the MSB which is 1

List Item + 16 contains the number of scan lines to read (image height)

List Item + 20 contains an offset to add to the hub memory address per scan line copied

List Item + 24 contains an offset to add to the external memory address per scan line copied

List Item + 28 contains a HUB address of the next request list item (or zero to end the list)

**Note \*:** Scan lines and Hub Buffer Size should not be zero.



## Graphics Write

For graphics image writes into external memory from HUB, the request list item format is this:

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/DstAddrHi (4 bits) + DstAddrLo (24 bits)
List Item+4		Hub Address
List Item+8		Hub Buffer Size*
List Item+12	MSB = 1	0
List Item+16		Scan Lines*
List Item+20		Scan line offset for DstAddr
List Item+24		Scan line offset for Hub Address
List Item+28		Link to next request

List Item + 0 represents the external memory destination address being written, and uses a write burst request with MSB = 1

List Item + 4 represents the HUB address to read the data from

List Item + 8 represents the number of bytes to write per scan line (controls image width)

List Item + 12 is zero apart from the MSB which is 1

List Item + 16 contains the number of scan lines to read (image height)

List Item + 20 contains an offset to add to the external memory address per scan line copied

List Item + 24 contains an offset to add to the hub memory address per scan line copied

List Item + 28 contains a HUB address of the next request list item (or zero to end the list)

**Note \*:** Scan lines and Hub Buffer Size should not be zero. HyperFlash cannot be the destination.

## Graphics Fill

For graphics fill operations into external RAM the request list item format is this:

HUB layout	Bit 31	Bits 30-0
List Item+0	MSB = 1	Req (3 bits) + Bank/DstAddrHi (4 bits) + DstAddrLo (24 bits)
List Item+4		Fill Data Pattern
List Item+8		Count*
List Item+12	MSB = 1	0
List Item+16		Scan Lines**
List Item+20		Scan line offset for DstAddr
List Item+24		0
List Item+28		Link to next request

List Item + 0 represents the external memory destination address being written to, and uses a write byte or word or long request with MSB = 1

List Item + 4 represents the fill data pattern to write to external memory

List Item + 8 represents the number of byte/word/long items to fill per scan line (e.g. pixels)

List Item + 12 is zero apart from the MSB which is 1

List Item + 16 contains the number of scan lines to fill (or zero for special handling\*\* below)

List Item + 20 contains an offset to add to the destination memory address per scan line filled

List Item + 24 contains zero

List Item + 28 contains a HUB address of the next request list item (or zero to end the list)

**Note \*:** Count should not be zero. HyperFlash cannot be the destination.

**Note \*\*:** if Scan Lines is set to zero, this optionally triggers a special graphics handler to execute. Once the first pixel or row of pixels is drawn, the code branches out to a handler that can determine what pixel addresses to write to next. The intent of this is to support arbitrary angled lines and other features. Complete implementation of this is TBD and may evolve more over time.

## Driver Initialization from PASM2 COGs

While typically started from the SPIN2 code, the PASM2 bus driver component(s) can still be spawned from a COG running PASM2 code when SPIN2 is not available. Any bus/device/address mapping and all other setup and management features normally provided by the SPIN2 API will then need to be handled by the PASM2 clients themselves in this case and is not covered here.

When the PASM2 bus driver is started it is also passed a pointer to an 8 long startup parameter structure held in HUB RAM. This is passed through PTR\_A at the time the driver COG is spawned. This HUB RAM structure contains the following information:

HUB layout	Parameter
StartupParams+0	P2 driver operating frequency in Hz, used for determining reset delays
StartupParams+4	Startup configuration option flags for the driver
StartupParams+8	P2 port A (lower 32 pins) reset mask of all devices on the bus
StartupParams+12	P2 port B (upper 32 pins) reset mask of all devices on the bus
StartupParams+16	(Address size << 24)   (Address bus base pin << 16)   Data bus pin
StartupParams+20	Pointer to 32 long device parameter structure in HUB RAM
StartupParams+24	Pointer to 8 long COG QoS parameter structure in HUB RAM
StartupParams+28	Mailbox base address for the driver to use in HUB RAM

The startup configuration option flag bits currently defined are:

Bit	DESCRIPTION
31	Set to 0 to use regular sysclk/2 transfer rate reads, or Set to 1 to optionally enable high speed sysclk/1 transfer rate reads
30	Set to 0 to for sysclk/2 transfer rate writes, or set to 1 for experimental sysclk/1 writes
29	Set to 0 to use normal registered clock pin timing, or Set to 1 to enable unregistered clock pin timing (for experimenting only)
28	Reserved for future use to enable further HUB-exec based graphics line/pixel drawing

The port reset masks in the startup data are used to identify which P2 pins on port A and B will be pulsed to a logic low level at driver startup, enabling optional memory device reset strobes in systems that use these. Setting a bit to 1 will enable a reset pulse on the corresponding P2 pin.

The base P2 pin numbers of the address and data buses are also provided along with the base address of the mailbox group this driver will use. Two remaining pointers are used for referencing the device and QoS data structures held in HUB RAM, and are defined in the next sections.

## Device Parameter Structure

This is arranged as 32 longs in total containing 16 bank parameter longs followed by 16 pin parameter longs, one long of each per bank in bank ID order. Devices over 16MB in size span multiple banks and require duplicate values configured in each associated bank's pair of longs.

The device and bank information along with the pin mapping is setup once at driver COG initialization time based on how the devices are mapped on the memory bus. Once the driver is up and running some of their fields can still be changed via some SPIN2 control methods, but no further memory devices on this bus can be created dynamically after this time without a driver restart.

Bank parameter long format:

Bits 31-16	Bits 15-12	Bits 11-8	Bits 7-0
Maximum Burst	Delay	Flags	Device size

Maximum Burst (16 bits):

This is the burst size allowed for the *device* in bytes, assuming a sysclk/1 transfer rate. Whenever the device transfer duration is unrestricted, (i.e. for HyperFlash) it can effectively be set to the P2 streamer transfer limit of just under 64kB bytes (it should be \$FFF0 for Flash page alignment purposes).

For HyperFlash banks, the 3 LSBs in this field are also shared to indicate which COG is currently allowed to modify the HyperFlash or access it's registers in an uninterruptible sequence whenever flag bit 11 is also set true.

For HyperRAM/PSRAM this burst field is normally configured to limit the chip select low time to be less than 4us/8us to avoid encountering refresh problems, but it can still be adjusted further.

- Delay (4 bits):

This nibble is comprised of two fields which create the input delay needed for external memory reads. These values vary with operating P2 frequency (and potentially temperature, PCB routing).

BIT	NAME	DESCRIPTION
15-13	DELAY	P2 clock delay cycles added while waiting for valid data input to be returned
12	UNREG	Set to 0 if bank uses registered data bus inputs for reads, Set to 1 if bank uses unregistered data bus inputs for reads

- Flags (4 bits):

The flag bits associated with each bank are shown below.

BIT	NAME	DESCRIPTION
11	PROT	Set to 0 if Flash bank is not exclusively protected by a COG Set to 1 if Flash bank is exclusively protected by a COG
10	TYPE	Set to 0 if bank is HyperRAM / writable memory Set to 1 if bank is HyperFlash / read only memory
9-8	RSVD	Reserved, set to 0

- Device size (8 bits):

This size field contains the number of address bits - 1 used by the device mapped into this bank. A sample of the mapping is shown. Any devices < 16MB still need to be on a 16MB boundary.

VALUE	DEVICE SIZE
23	16MB
24	32MB
25	64MB
26	128MB
27	256MB

**NOTE:** If the bank is not in use all of the bank parameters in the long MUST remain zeroed.

The 16 bank parameters identified above are then followed by the 16 pin parameter longs (one for each bank). The parameter format depends on the particular type of memory in use.

The pin parameters for HyperRAM and HyperFlash devices use this format:

Bit 31	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Invalid	Latency Edges	RWDS Pin	CLOCK Pin	CS Pin

The Latency Edges field is set to twice the number of latency clocks configured for this device.

The three pin fields select which P2 pins connect to the device's control pins (from 0-63). All control pins are required to be defined and are used by this driver. The clock pin can be shared by different devices on the same bus.

The pin parameters for PSRAM devices use this format:

Bit 31	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Invalid	Reserved	Reserved	CLOCK Pin	CE Pin

The two pin fields select which P2 pins connect to the device's control pins (from 0-63). Both control pins are required to be defined and are used by this driver. The clock pin can optionally be shared by multiple devices on the same bus.

The pin parameters for asynchronous SRAM devices use this format:

Bit 31	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Invalid	Reserved	WE Pin	OE Pin	CE Pin

The two pin fields select which P2 pins connect to the device's control pins (from 0-63). All three control pins are required to be defined and are used by this driver. The OE and WE pins can optionally be shared by different devices on the same bus.

The pin parameters for HUB RAM "devices" (as simulated external memory) use this format:

Bit 31	Bits 31-20	Bits 19-0
Invalid	Reserved	Hub RAM base address

HUB RAM is somewhat different to the external memory devices and there are no pin fields in this long, but rather a base address identifying where this bank is situated in HUB RAM. The only purpose of this memory type is for testing requests made to the driver and debugging software without requiring real external memory to be present. It would not be used in real applications as you would simply read the HUB memory directly for higher performance without the service latency overhead (and COG use) of the driver.

In all cases above, if a bank is not in use, bit 31, its Invalid bit, must be set to 1, otherwise if the bank is valid it should be cleared to 0.

## COG QoS Parameter Structure

The COG QoS parameter settings are maintained in another 8 long structure in HUB RAM (one long per COG, in COG ID order) using the following data format per long:

Bits 31-16	Bits 15-12	Bits 11-10	Bits 9-0
Burst Limit	Priority	Flags	Internal use

Configuring this QoS structure is the key to having COGs share the external memory in a controlled way, and to optionally ensure that some COGs will be treated preferentially over others where required. Real time driver COGs can be configured with a higher priority over normal COGs for example so their memory transfer requests can be repeatedly issued and serviced without being noticeably impacted by other COGs. Lower priority COGs can still get access to memory but can be held off until higher priority COGs have had their requests serviced first.

- Burst Limit (16 bits):

This restricts a COG's maximum burst size before fragmenting which helps bound the latency for servicing the highest priority COG after the lower priority COG request. A smaller burst size setup for lower priority COGs would let a video COG have its requests serviced sooner, for example. This burst size is measured in bytes.

To guarantee that real-time (e.g. video / audio) transfers can be reliably sustained without artefacts, the burst size for all lower priority and round robin COGs should be set to a value that allows both the real-time COG's transfer and any single lower priority (or round-robin) serviced COG's transfer to be completed before the next real-time request requires processing.

In the specific case of video, both the video COG and any lower priority COG's transfers will need to complete in a single scan line (with some margin for overheads). In this case the optimal burst limit will depend on the video timing, the resolution and colour depth as well as the P2 frequency and actual transfer rate on the Hyper bus.

So if you had a 31us video scan line for example, and if the video burst transfers per line totals 24us after any fragmentation/overheads, no more than 7us of bursts (including overheads) should be allocated to any of the COGs lower in priority. That way, once any lower priority COG is serviced after a video COG's transfer, it's request will complete in time such that the next video request can occur and complete again before the total deadline expires.

This burst limit per COG works in conjunction with the device's own burst size. The lesser of the two values will apply to the burst transfer sizes on the bus and determines the number of bytes sent before transfer fragmentation occurs. Fragmentation also provides an opportunity for higher priority COGs to be serviced. Importantly, to avoid internal page boundary crossing problems if HyperFlash is used, the COGs that access Flash need to set their burst limit to a multiple of 16.

- Priority (4 bits):

This field holds a priority polling flag and the 3 bit priority assigned to the COG being serviced by the driver. Strict priority COGs are serviced before any round-robin COGs. Round-robin polled COGs try to share the bandwidth remaining “fairly”, by request but not necessarily by bandwidth.

PRIORITY	DESCRIPTION
%1_111	Highest priority polled COG
%1_110	Second highest priority polled COG
%1_...	...
%1_000	Lowest priority polled COG (but still above round-robin COGs)
%0_xxx	Round-robin COGs, for xxx<>0 they fail with ERR_BUSY if accessing locked flash
%0_000	If xxx equals 000, the RR COG stalls if accessing flash locked by another COG

- Flags (2 bits):

These per COG flags allow selection of the notification method, as well as the ability to prevent the COG's memory access from being pre-empted by other COGs when fragmentation occurs. This is useful for a couple of reasons:

- 1) performance - avoiding COG polling during fragmentation reduces software overheads and increases the bus utilization
- 2) data integrity - allows atomic transfers to/from memory if larger structure in memory need to remain consistent before other COGs access or modify the data.

Flag bits	NAME	DESCRIPTION
11	ATN NOTIFY	if 0, serviced COG is notified of completion/error via its mailbox only if 1, serviced COG is notified with ATN on completion as well as in its mailbox
10	LOCKED BURST	if 0, COG can be pre-empted if its burst is fragmented (polling restarts then) if 1, COG's transfer continues to completion before further polling

The 10 remaining least significant bits in QoS Parameter long are ignored when writing them to the driver. These are used for internal purposes in the driver.

This per COG QoS structure in HUB RAM can be modified and re-loaded into the driver if parameters need to be changed at run-time. Use the *setupQos* SPIN2 method to update this structure or alternatively issue a request with type %111 directly to the control mailbox from PASM2 clients.



## Current Limitations

There are several limitations to consider when using HyperRAM / HyperFlash and this driver.

- In all cases with memory fills, copies and burst read/write operations, if the number of transfers from the starting source address (or destination address) exceeds the last address for that device's size, the address will just wrap around within the device. Both the source and the destination address increment only, so reverse copies are not possible.
- Writes at Sysclk/1 transfer speeds are only experimentally supported by this driver and may require additional HW support to delay the clock signal in order to get this working reliably.
- Due to the way the P2 streamer samples the input data, there is a chance that incoming read data from external memory devices will not be stable at the exact instant it is clocked into the P2 for all frequency values. There is an internal delay parameter that allows read timing to be adjusted to try to compensate for this issue, and some experiments with the P2-EVAL and the HyperRAM/HyperFlash breakout module found that over the operating P2 frequency range at room temperature the memory devices typically have some amount of overlap between 1-2 or 3-4 working delay values depending on transfer speed and the P2 clock rate where input data is stable enough to sample, however this could change with different board layout and temperature ranges, creating different delay profiles over frequency. A method exists to change the frequency/delay mapping profile in use (on a per device basis), and in advanced cases where reliability is even more critical it could even be linked to current operating temperature if this delay/frequency/temperature relationship is already measured for a given setup. Some process could monitor temperature and/or adaptively update the driver in an attempt to compensate for delay variation. More experiments would likely be needed to see just how much temperature variation exists at the particular operating frequencies of interest.
- Currently only version 1 HyperRAM has been tested with this driver. It is anticipated that version 2 HyperRAM should ultimately work too (with minor updates to latency), but until actual device hardware becomes available it is not yet possible to test to confirm there are no issues with the newer devices. Determining which type of RAM is fitted will also be required.
- In some MCP (multi-chip package) HyperRAM devices, burst transfers cannot be made to work contiguously across the die boundaries because the burst simply wraps inside the dies within the package. It is important to keep this in mind when laying out any large structures in memory such as frame buffers, which may need to be maintained in one die or the other.
- HyperFlash read burst transfers that cross page boundaries can have issues if some care is not taken, causing extra unwanted bytes to be inserted in the middle of the read transfers. The driver has a workaround dealing with this but still depends on the lesser value of the configured burst sizes for the HyperFlash device and for the COG's accessing it to always remain a multiple of 16 bytes. To avoid problems, you can leave the HyperFlash bank's burst size as its default, and set the COG's burst limit to less than \$FFF0 and aligned to 16 bytes.

- HyperFlash reads are aligned to their data access size just like the P1 memory reads were. Reading a long at byte offsets 1, 2, 3 from a long aligned address will return the same value as found at the aligned address offset 0, (i.e. the 2 address LSBs get interpreted as zero) when reading longs. Similarly reading words at odd HyperFlash addresses will return the same word that is read at even HyperFlash addresses.
- HyperRAM reads and writes are unaligned and behave like P2 memory accesses so words and longs can be read from any byte address and will return the next 2 or 4 bytes from that initial location respectively.
- For the 8 bit asynchronous SRAM devices supported by this driver, their data bus must be situated on an 8 bit group, and the address bus pins cannot span across the 32 bit port A, B boundary. Also it makes sense to keep the control and data pins of a port *below* the address pins (or on a different port) because the address bits are incremented in the port pin registers directly and if they wrap around sufficiently during long burst transfers, the address MSB may overflow into a control pin or data pin on the same port. A safe SRAM pin mapping on a single P2 port is to have the 8 bit data first at lowest pin numbers, followed by control pins, followed by address bits at higher pin numbers. This then supports up to 2MB SRAM devices using 32 pins which at this stage is about the largest SRAM you can obtain at a realistic price. Any larger addressed devices would need to place the control or data pins on a different port.

## Future Directions

Further work may occur over time to add support for other memory devices such as QSPI Flash (e.g. Winbond W25Q128 devices typically used by the P2 as Boot flash), HyperRAM v2? It could also be interesting to find a way to memory map SD card volumes as block devices in order to use them as custom file systems. The existing 256MB address size limit per bus might be increased by addressing 512 byte sectors or other block sizes in that case.

Now that more memory types are being supported and this code has evolved beyond supporting just HyperRAM/HyperFlash, ideally there would be some way to repartition the driver source over different files to help remove unnecessary code making it into the driver. E.g., including all the SPIN2 code from HyperFlash writing methods is undesirable whenever that memory type is not used. FlexSpin already removes dead code for us, but official SPIN2 current does not, leading to unnecessary image size increases when the full set of these APIs are always included in the code.

Some type of adaptive and automatic fine tuning of the required device read delay could be useful to compensate for temperature changes affecting the board's timing over time. This may require an occasional call from a non-driver COG to invoke this tuning or some other method.

Further graphics capabilities might be added to the driver to speed up some drawing operations.

If the APIs can be somehow simplified further it requires less work for the client, so that could be investigated. Optimizing the burst size setting using calculations in the driver would be useful.