MBus M3 Implementation Specification

Pat Pannuto <ppannuto@umich.edu>

Revision 0.3* — May 22, 2013

This document details the implementation of MBus for the M3 project. Other devices compatible with MBus should not require any information from this document. This document covers the hardware design of the M3 MBus implementation.

Contents

1 Design Decisions

1.1 Pending Signal

Stream-Oriented versus Datagram-Oriented

The previous (I2C) bus implementation keeps an internal queue of address+word pairs that need to be sent on the bus. Once a transaction begins, the processor could fill the queue one word at a time and if another word arrives before the previous word has finished transmitting the new word is appended to the end of the current transmission. The result is a *stream*-style interface, which requires message boundaries (if desired) to be implemented in software, which introduces overhead such as separate acknowledgment *messages* that must be sent over the bus from the receiving node of a transmission.

With MBus, we choose instead to build a *datagram*-style protocol. In particular, this provides a very important primitive to the programmer: One request to send one message is received in whole or not at all by the intended recipient. As a receiving node generates an interrupt when a MBus transaction received inviting the recipient to read out the whole received message, it is important for this guarantee that each request to send a message generate one and exactly one transaction on the MBus.

In the hardware implementation, this means a queue of bytes that (possibly) non-deterministically merges messages will not work to uphold this contract. Instead, we design the interface with an explicit "Pending" signal that clearly delineates the boundaries between messages.

1.2 Interrupts

The two traditional interfaces are either a single multiplexed interrupt where the handler reads a status register or a unique interrupt per event. Unique interrupts are cheap to implement. Additionally interrupts take up space in the vector table, but the same amount of space plus overhead would be consumed by the jump table of the interrupt demultiplexing software implementation anyway.

Given this, the M3 implementation uses a unique interrupt per event.

1.3 Transaction architecture and DMA

The interface provided by the new bus is essentially a DMA-style interface. Ideally this should allow for a large amount of logic re-use. In particular, external DMA transactions look just like normal transactions, except the destination of the bus transaction is controlled by the transmitter instead of selected by the receiver.

2 RTL Design

The M3 MBus implementation defines two major components: a *Bus Controller* and a *Layer Controller*. The bus controller understands the MBus protocol and presents a simple word-wide interface to higher layers. The generic layer controller provides a register file and a memory interface, sufficient for most simple devices. The M3 CPU layer replaces this block with a custom, more complex structure. In addition, the M3 MBus implementation requires some support blocks: a *Sleep Controller*, a *Wire Controller*, and an *Interrupt Controller*. The are hand-optimized always-on components designed for minimal power draw. This design is depicted in Figure 1:

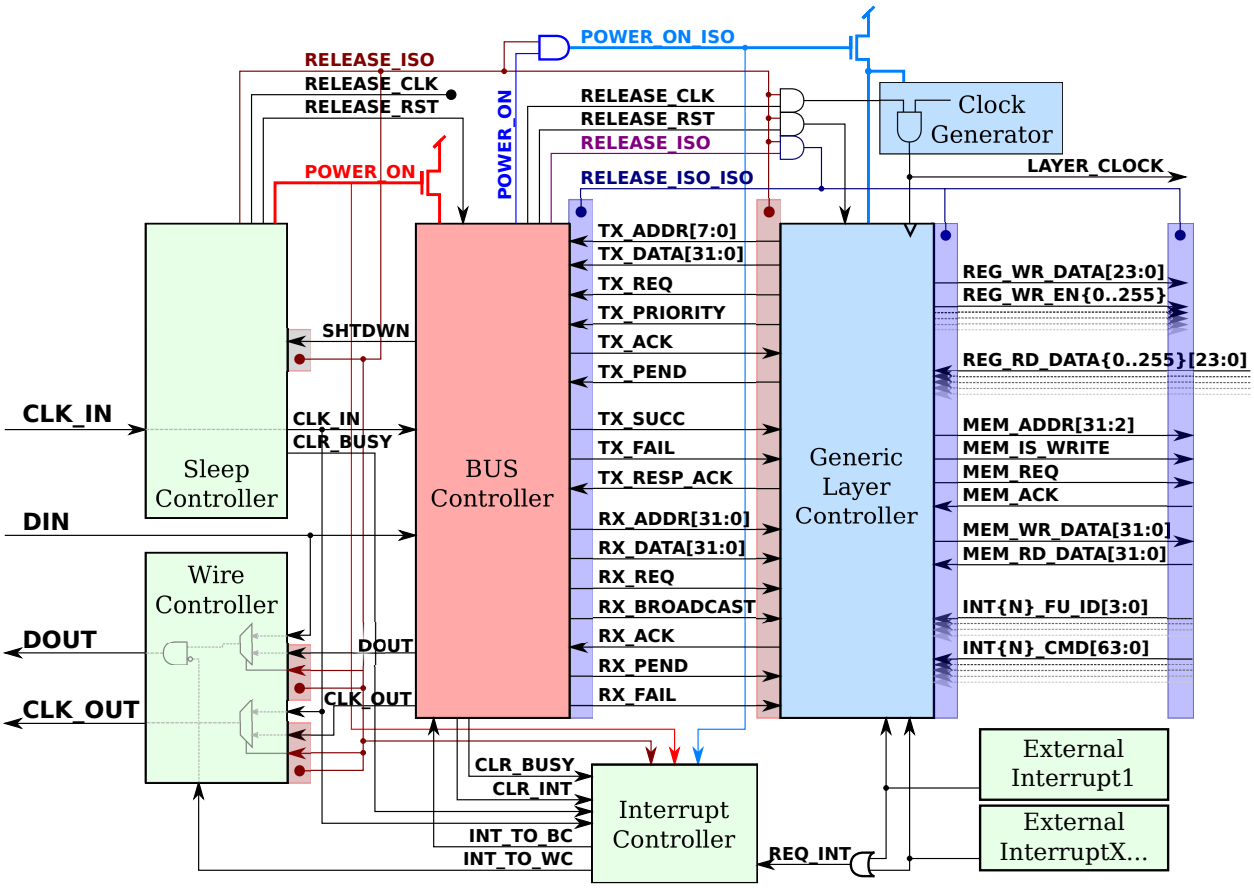


Figure 1: *Implementation block diagram*— The green blocks are always-on components. The two power-gated power domains are shown in red (Bus Controller) and blue (Layer Controller). Module I/O isolation is indicated by the large rectangles controlled by `ISO(LATE)` signals. Circuits elements presented are for conceptual understanding not final design of glitch-free logic. The global reset signal is omitted for clarity.

The four standard power control signals are:

Signal Name	Function	Power-Up	Power-Down
`POWER_ON`	Controls Power-Gating	1st	2nd
`RELEASE_CLK`	Supply Clock to Internal Logic	2nd	2nd
`RELEASE_ISO`	Electrically Isolate Module I/O	3rd	1st
`RELEASE_RST`	(De)Assert Reset	4th	2nd

The `RELEASE_CLK` signal may be omitted if the power-gated layer does not have an internal clock.

2.1 Module Block Details

This section provides details on the inner workings of each of the supplied modules.

2.1.1 Sleep Controller

The job of the sleep controller is to wake the bus controller when a message on the bus begins. The sleep controller is also responsible for powering down the bus controller when requested, using edges from MBus to do so. For layers capable of waking on events other than MBus transactions, the sleep controller may be extended or need to coordinate with other sleep controllers. This document assumes only a basic sleep controller.

Signals

Power Signals The sleep controller uses both edges of the `CLK_IN` signal to generate the power signals. The rising edge that resolves arbitration is used to release the power gating. The falling edge (Priority Drive) is unused and reserved for implementations that require a `RELEASE_CLK` signal. The next rising edge (Priority Latch) drives `RELEASE_RST` and the falling edge (Begin Transmission) drives `RELEASE_ISO`. The next rising edge latches Bit 0 of the transmission.

MBus Signals The sleep controller samples and passes through the MBus `CLK_IN` signal. The sleep controller uses this line to clock its internal state machine. The sleep controller and the bus controller can be safely considered synchronous modules with respect to one another as they both rely on the MBus`CLK_IN` signal to provide a clock.

Module Signals The sleep controller has only one non-reset input signal: `SHTDWN`. This signal is asserted by the bus controller to request that it be put to sleep. This signal is sampled synchronously with the MBus clock. The `SHTDWN` signal may only be asserted during the falling edge Drive Control Bit 1.

The sleep controller is also responsible for generating a `CLR_BUSY` output signal when it powers off the bus controller. This signal is normally generated by the bus controller at the end of MBus transactions and is used to indicated to the interrupt controller that the bus is no longer busy.

2.1.2 Wire Controller

The wire controller is responsible for physically driving the MBus wires. This separation is necessary to ensure that the data and clock lines are forwarded even when the bus controller is power gated. The wire controller is very simple module, conceptually it is little more than a mux that either forwards the values coming in from the bus or values coming from the bus controller. The wire controller also must allow for the interrupt controller to induce a glitch on the bus for wakeup as discussed in the Power Design section of the *MBus Specification*.

Changing the wire controller muxes requires extreme care to ensure that glitches are not introduced. In practice, this means only changing the clock mux signal when `CLK_IN` is already high. The data mux is more complicated. During Idle, the data line is high, any glitches while pulling it low (so long as it ultimately is held low) will be significantly shorter than t_{long} and therefore ignored. During transmission, the state of the data line is not always known when control transitions are required. MBus ignores the data line while the clock is low. Any potentially glitch-inducing changes to the data mux must occur on the falling edge of clock to avoid uncertainty.

The wire controller also facilitates external interrupts as a wakeup source via a *glitch inducer*. The goal of the glitch inducer circuit is to request a bus arbitration cycle and *lose* it. Nominally, this results in no winner of arbitration and the bus will quickly reset. However, the glitch inducer logic must also correctly handle the case where another node was performing genuine arbitration at the same time and a real message takes place.

2.1.3 Interrupt Controller

The interrupt controller is responsible for directing interrupt events to the appropriate modules at the appropriate times. The destination of an interrupt event depends on the current power state of the bus controller and layer controller.

If the layer controller is powered on, the interrupt controller blocks the interrupt signal as the layer controller will handle the interrupt and generate any appropriate messages. If the layer controller is powered off, however, the bus controller must wake the layer controller. To do this, the bus controller must harvest edges from MBus. To generate the power signals, the bus controller must observe a transaction while the `INT_TO_BC` signal is asserted. To generate these edges, the interrupt controller must induce a glitch.

A Subtle Detail: Extreme care must be taken when this glitch is induced, in particular it is important to ensure that MBus is idle so that a real glitch is not created. The simple approach would be to ask the bus controller to simply export a "bus busy" signal, asserted while a transmission is active. Such a signal is not sufficient, however. If a bus controller is not powered on, it is incapable of asserting busy. If two nodes have interrupts near each other in time, the first node will induce a glitch unobserved by the second node. If an interrupt occurs on the second node between end of t_{long} and resultant falling edge of `CLK_IN` and the rising edge for arbitration, the second node will unintentionally and worse unknowingly win arbitration. As the node does not know that it has won arbitration, it will never end the transmission and the MBus will be locked until the master node's watchdog counter expires.

To avert this issue, the interrupt controller keeps an internal sense of "bus busy". The interrupt controller latches the bus as busy whenever `CLK_IN` goes low. The internal busy signal is then cleared by an explicit signal from the bus controller at the end of MBus transactions. As a special case, the sleep controller must clear the busy status when it powers the bus controller down as the bus controller will be powered off at the end of the transaction and incapable of generating the signal. This clear busy signal is internally treated as the last step in the power-down sequence.

2.1.4 Bus Controller

The bus controller is responsible for handling all possible bus events, including generating acknowledgments if it is the target of a transaction. The bus controller is only obligated to hold one word of a transaction, devices wishing to receive messages longer than one word in length must have a local FIFO to store partial messages in their layer controller.

Signals

Power Signals The bus controller provides the same power control signals to the layer controller that the sleep controller provided to the bus controller. For basic layers, the bus controller is responsible for waking and sleeping the layer controller.

The bus controller should only wake the layer controller if there is a message of *non-zero* length being transmitted to the layer's address. In practice this means the bus controller should not begin waking the layer controller until it has received the *third* data bit. Layer controllers should **not** be awoken for broadcast messages.

Once the bus controller elects to begin waking the layer controller, it **must** wake the layer controller completely. Even if another data bit is never sent (Interrupt) after the bus controller elects to wake the layer controller, the bus controller still has the Begin Interrupt, Control Bit 0, Control Bit 1, and Begin Idle edges to use to wake the layer controller.

The bus controller will indicate an erroneous wake-up by asserting `RX_FAIL` signal. The layer controller must acknowledge (`RX_ACK`) the erroneous transmission before the bus controller is capable of receiving another message. This enables the layer controller to take action to put itself back to sleep after a spurious wakeup.

MBus Signals While the bus controller is not directly connected to the external pads for all MBus signals, it does use all MBus inputs:

INPUT	CLK_IN	Bus Clock In
OUTPUT	CLK_OUT	Bus Clock Out
INPUT	DIN	Bus Data In
OUTPUT	DOUT	Bus Data Out

Module Signals

INPUT	RESET	Global system reset
OUTPUT	SHTDWN	Request power gating
INPUT	TX_ADDR[7:0]	Address to transmit
INPUT	TX_DATA[31:0]	Data to transmit
INPUT	TX_REQ	Request to transmit
INPUT	TX_PRIORITY	Is high priority message?
OUTPUT	TX_ACK	Acknowledge request to transmit
INPUT	TX_PEND	More data pending
OUTPUT	TX_SUCC	Transmit successful
OUTPUT	TX_FAIL	Transmit failed
INPUT	TX_RESP_ACK	Acknowledge TX successful / fail
OUTPUT	RX_ADDR[31:0]	Destination address of RX'd packet
OUTPUT	RX_DATA[31:0]	Data received
OUTPUT	RX_REQ	Data is ready
OUTPUT	RX_BROADCAST	RX'd message was a broadcast message
INPUT	RX_ACK	RX_DATA has been saved
OUTPUT	RX_PEND	More data is coming
OUTPUT	RX_FAIL	Abort current RX

The bus controller provides a single-word interface to MBus to higher level modules. To ensure that the bus controller can send a timely ACK for a received message, modules are obligated to be able to (eventually) receive a minimum of one word. That is, if the RX_REQ line is raised, the module *must* eventually receive that word by signaling RX_ACK. If another message is received while RX_REQ is still high, the bus controller will Interrupt the bus indicating "01" (~EoM, TX/RX Error). This serves to (i) save bus bandwidth by canceling a message that will not be received, and (ii) indicate to the transmitting layer that there is still a pending message (or message part) in the receiving layer.

If the TX_PEND signal is asserted but a TX_REQ for a new word is not sent by the time the first word has been sent, the bus controller will Interrupt the bus indicating "01" (~EoM, TX/RX Error). The bus controller will also assert TX_FAIL, at which point the transaction must be aborted.

The RX_PEND signal requires special attention. When the RX_REQ signal rises, if RX_PEND is also high, it indicates that there is more data to follow. If a module asserts RX_ACK in response it is obligating itself to receive at least one more word after the current word[1]. If a module cannot receive another word beyond the word it is currently latching, it should ignore RX_REQ. When the next word is received by the bus controller, it will detect that RX_REQ is still high and abort the entire transaction, Interrupting to indicate RX Error. In practice this means a transmitting node may believe it has sent one more word than was actually received. As the entire transaction is NAK'd however, the only implications are for the software flow control estimation, which can easily compensate.

For simple nodes that only support single word transactions, this RX_PEND subtlety is important. Such a node should wire its final acknowledgment output signal in a manner such as

```
TX_ACK_out = (RX_PEND_in) ?  1'b0 :  internal_ack_signal;
```

to ensure it does not attempt receipt of multi-word transactions. The TX_PEND signal can simply be tied low.

[1] This is logically a continuation of the original contract. In an idle state, a module has received zero words thus far and is obligated to be able to receive one more. If a module ACKs a word while RX_PEND is high, it is accepting the current word *and* committing to receive the next pending word.

Parameters The bus controller requires relatively little configuration. The only available parameter is the address(es) that this bus instance should respond to:

`ADDRESS`	Address(es) to receive and acknowledge
`ADDRESS_MASK`	Which bits of `ADDRESS` are significant

2.1.5 Generic Layer Controller

The layer controller is responsible for communicating with the bus controller and facilitating multi-word transactions. It federates access to the individual components on a MBus member node. Most M3 chips use the common Generic Layer Controller, described here. The CPU, however, requires the more complicated CPU Layer Controller.

Power Signals The generic layer controller can only be woken by the bus controller. Layers capable of generating an interrupt while M3 is in sleep mode (e.g. an alarm) must wire into the glitch inducer via the interrupt controller to wake the layer and the whole M3 system. The generic layer controller does not output any power signals.

There is no explicit shutdown signal from the layer controller to the bus controller. Rather, the layer controller issues a broadcast message announcing to the bus its intention to shut down. The bus controller will recognize the special broadcast message and shut down the layer controller once it is sent.

Module Signals (Bus Side) All layer controllers have a common set of signals to interface with the bus controller, as shown in Figure 1. Details of these signals are presented in the bus controller documentation: 2.1.4 Module Signals.

Module Signals (Interface Side) The generic layer controller expects to communicate with two types of hardware: a register file and a basic memory. One or both of these may be instantiated.

Register File: The register file is for small, simple configuration, status, or action bits. It presents an 8 bit address space with 24 bit wide data. Writes are pulse-triggered, with a unique write control line for each of the registers. Data to read must always be valid, with no indication that data is being read.

Registers may be smaller than 24 bits. Unused bits for writing should be left disconnected. Unused bits for reading **must** be tied **low**.

Memory: The memory interface is for larger amounts of data (images, audio, packets, etc). Memory is a 32 bit address space with 32 bit data. The memory space does not alias the register file address space.

The memory interface is a simple two-wire handshake that indicates a request when signals are valid and an acknowledgment when the data has been latched. Memory must be fast enough that it can support streaming reads / writes at the line speed of MBus.

2.1.6 CPU Layer Controller

TODO: Document CPU Layer Controller. Consider separate CPU impl document?

2.2 Module Interdependencies

In addition to the module signals, the following additional requirements must be considered for the blocks to operate correctly.

2.2.1 Clock Domains

The bus controller and layer controller are on separate clock domains. Care must be taken with the `REQ` and `ACK` two-wire handshake to ensure that signals are all double-latched. Double latches should not be required for the other signals are they are only sampled after the `REQ` or `ACK` line is stable.

The generic layer controller does not perform double-latching with any of the signals attached to the register file or memory. These blocks are expected to run on the same clock as the layer controller. Designs that violate this assumption must make modifications to ensure signal stability.

2.2.2 Relative Clock Frequencies

Nominally, the layer controller would only require activity once every 32 MBus clock cycles. In practice, however, there is a double-latched, bidirectional handshake between the bus controller and layer controller. At a minimum then, the layer controller must be at least $1/8$ the speed of the bus.

This timing constraint further extends to the memory if longer bus transactions are going to be supported. The generic layer controller is a write-through device, it does not buffer memory requests. Consider the memory latency in cycles M to be defined as the maximum possible number of cycles between the assertion of `MEM_REQ` before the `MEM_ACK` response is asserted, then the minimum clock speed of the layer controller and memory is $\frac{6+M}{32}$ of the bus speed.

There is no upper bound on the layer speed relative to the bus. Layer controller clock speeds are generally configurable. See M3 Register/Memory Maps for details.

3 M3 Addressing and Messages

The MBus addressing attempts to minimize change from the legacy M3 implementation.

3.1 Full Prefix Assignments

The old I2C layer IDs are mapped to the bottom of the full prefix address space:

Legacy Layer ID	Full Prefix	Layer Name
0b1000	0x00008	CTR Layer (Registers)
0b1001	0x00009	DSP Layer
0b1011	0x0000b	UWB / RADIO Layer
0b1100	0x0000c	IT Layer (Imager)
0b1101	0x0000d	IT Layer (Timer)

3.2 M3 Sleep / Wake Rules

A layer controller enters sleep mode only when explicitly requested to by a MBus Sleep message. A layer controller will be woken for any message that explicitly targets that layer.

Any broadcast message will wake the layer controller of the CTR layer. Broadcast messages will not wake any other layer controller.

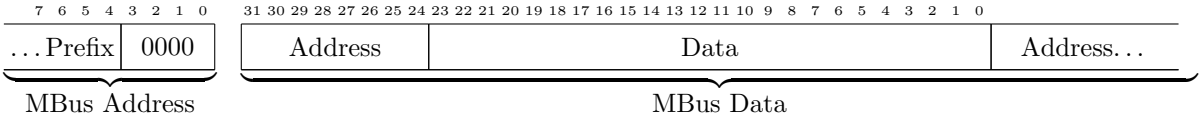
3.3 Functional Unit Addressing / Messages

M3 layers are defined to have two completely isolated regions: a *register space* and a *memory space*. Layers may have either or both of these constructs.

3.3.1 Register Space

The M3 register space is an 8 bit addressable array of 24 bit wide registers. This register space is generally sparsely populated. Any undefined bits are treated as RZWI (Read as Zero, Write Ignored). The register space contents are defined in M3 Register/Memory Maps.

Register Write

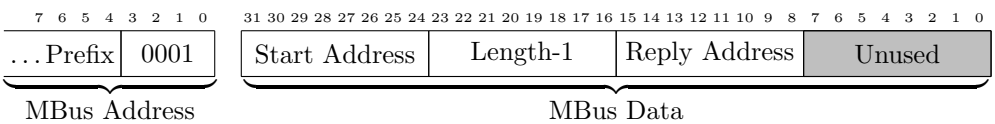


Bits 0-23 of the MBus data field are written to the register addressed by bits 24-31. The write occurs immediately, as soon as the layer controller receives the message. Multiple registers may be written in a single MBus transaction by sending multiple data packets. Each 32 bit chunk of data is treated as if it were an independent transaction.

Interrupt Semantics: Each command is applied immediately when it is received. A four-command message would have a $4 \times 32 = 128$ bit data payload. If 63 bits are received prior to interrupt, only the first command was applied. If 64 bits are received, the first two commands were applied.

TODO: Work through this from transmitter perspective around the full loop with various interrupters, make sure we can guarantee the TX node knows whether commands were applied.

Register Read



Bits 24-31 specify the address of the register to be read. Bits 16-23 may be used to request that multiple registers are sent. This field is a count of the number of values to be sent less one, that is a value of 0 requests 1 register is read and a value of 255 requests that all 256 registers are sent. Bits 8-15 are the address the reply is sent to. Bits 0-7 are unused and may be omitted on the wire. If sent, their value is ignored.

The response message is sent to the address specified in bits 8-15 of the request and its data field is formatted exactly as the Register Write command: 8 bit address + 24 bit data. This feature permits, *but does not require*, the layer controller to skip unpopulated addresses in the register space while replying.

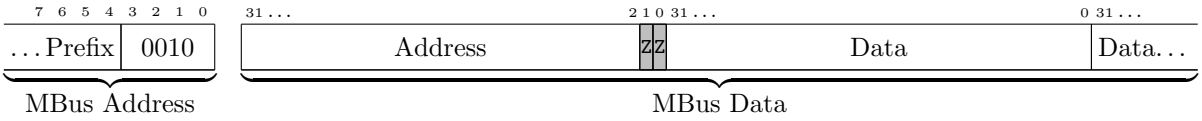
If the starting address field and subsequent length exceed the register space, that is a request for register #256 would have been made, the layer controller stops the current transaction after sending register #255.

Interrupt Semantics: If an interrupt occurs during reply, the transaction is aborted and is **not** retried.

3.3.2 Memory Space

The M3 memory space is a 32 bit addressable array of 32 bit words of memory. This memory space is generally contiguous. Any undefined accesses are treated as RZWI (Read as Zero, Write Ignored). The memory space contents are defined in M3 Register/Memory Maps.

Memory Write

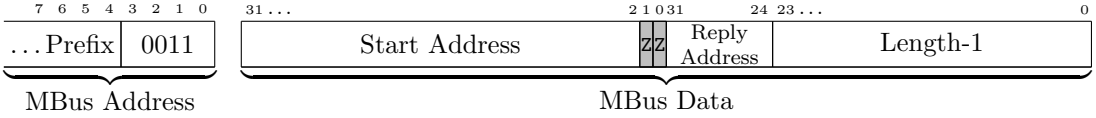


The first word received is the address in memory to begin writing to. The bottom two bits of the address field are ignored and **must** be transmitted as 0. Subsequent words are treated as data. The first word of data is written to the specified address. The next word of data is written to address+4 (the next word in memory) and so on. There is no limit on the length of this message.

Interrupt Semantics: Each word of data is written to memory immediately when it is received. A four-word message would have a $(1 + 4) \times 32 = 160$ bit data payload. If 95 bits are received prior to interrupt, only the first word was written to memory. If 96 bits are received, the first two words were written to memory.

TODO: Work through this from transmitter perspective around the full loop with various interrupters, make sure we can guarantee the TX node knows how many words were written.

Memory Read



The first word received is the address in memory to read from. The bottom two bits of the address field are ignored and **must** be transmitted as 0. The second word indicates the address to reply to and the length of the requested read *in words* less one. A length of 0 will reply with 1 word of data. A maximum length will request a read of 2^{24} words (2^{26} bytes, 64 MB).

The response message is sent to the address indicated in bits 24-31 of the second word of the request. The message is formatted exactly as the Memory Write command, that is a 32 bit address field followed by a series of sequential 32 bit data fields. In this way, layer A could request a DMA read from layer B that results in a DMA write to layer C.

If the starting address field and subsequent length exceed the memory space, that is a request for address `0x100000000` would have been made during the response, the layer controller stops the current transaction after sending address `0xfffffffc`.

Interrupt Semantics: If an interrupt occurs during reply, the transaction is aborted and is **not** retried.

3.4 Broadcast Channel 3: Member Node Events

The generic layer controller supports *Member Node Level Interrupts* only. The assignment of interrupt indicies and the messages used to clear interrupt are layer-specific and indicated in M3 Register/Memory Maps.

3.5 Deviations from MBus Specification

Details of MBus messages can be found in the MBus specification. To simplify the initial M3 implementation of MBus, some pieces were not implemented exactly to spec. These deviations are listed here:

3.5.1 Short Address Assignment

The CTR Layer does not participate in enumeration. It fixes its short address as `0b0001`. Enumeration is performed by the CPU on bootup and must begin at `0b0010` instead.

3.5.2 Messages Not Implemented

The following messages are not implemented and will be ignored:

- Channel 1:
  - Selective Wake By Short Prefix
  - Selective Wake by Full Prefix
- Channel 2:
  - There are no defined Channel 2 messages as of this implementation. Channel 2 is thus ignored.
- Channel 7:
  - All Channel 7 messages are ignored.

4 Programmer's Model

The MBus implementation provides the following primitives to the programmer:

synchronous	Configuration settings
asynchronous	Send single word message to address.
asynchronous	Send arbitrary-length message to address.
synchronous	Request number of words sent. (*synchronizing operation*)
asynchronous	Dropped message notification.
asynchronous	Notification of received message and type.
synchronous	Retrieve received single-word message.
synchronous	Retrieve received multi-word message length and address.

In the following sections, there are many reserved bits. These bits shall be considered RZWI (Read as Zero, Write Ignored). These bits are currently unused but are reserved for possible future use. While the value during writes are currently ignored, programs should write `0` to ensure future compatibility.

4.1 Memory Map

MBus claims the region of memory `0xA5000000-0xA6000000`. The next most significant byte is used to identify the type of request. The remaining five bytes are defined by each request type:

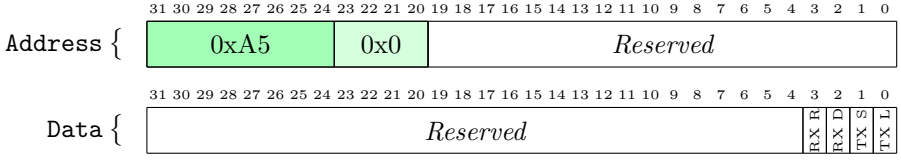
`0xA50`	`RW`	Interrupt Configuration [Read/Write]
`0xA51`	`W`	Single Word Write [Write Only]
`0xA52`	`W`	Arbitrarily Long Write [Write Only]
`0xA53`	`R`	Transaction Result [Read Only]
`0xA54`	`R`	RX Word (single-word) [Read Only]
`0xA55`	`RW`	RX Buffer (multi-word) [Read/Write]
⋮		*Reserved*

4.2 Configurable Parameters

4.2.1 Interrupt Configuration [Read/Write]

Depending on the system design, it may be desirable for an interrupt to fire when a transmission request is completed. Given the very different nature of long and short requests, they are configured separately.
There is no need to explicitly enable/disable receipt interrupts, as they are implicitly disabled if all buffer target written bits are set.

Bits Required	Purpose {Abbrv}
Address	
8	MBus Memory Map Location
4	Command Specifier
Data	
1	Long TX Complete Interrupt {TX L}
1	Short TX Complete Interrupt {TX S}
1	Bus dropped an RX 4.4.1 {RX D}
1	Bus received a message 4.4.1 {RX R}



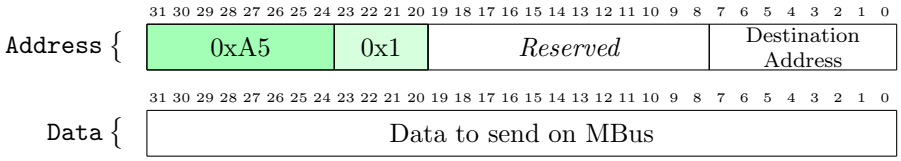
4.3 TX Transaction Registers

A properly written program will only issue one transaction at a time—that is, every request to send a message on the bus will be followed read of the success of that transaction before issuing another one. If a program does issue two requests back-to-back, the layer controller should block the second request until the first is completed. The second request should unblock the memory bus as soon at the first request is complete and the layer controller is ready to begin processing the second request. A request for the last transaction result should reflect the second transaction, blocking if necessary. There is no method for such an erroneously coded program to check the return status of the first message.

4.3.1 Single Word Write [Write Only]

As a small optimization for the simple, single-word case a single memory write can express all of the needed information without requiring the layer controller to issue another request to memory to retrieve data.
As there is no hardware retry primitive, the software is still obligated to issue a read to determine if the message send was successful. For similar reasons, this message is an asynchronous message and should return control to the processor as soon as the layer controller accepts the request, not waiting for the bus controller to send the message.

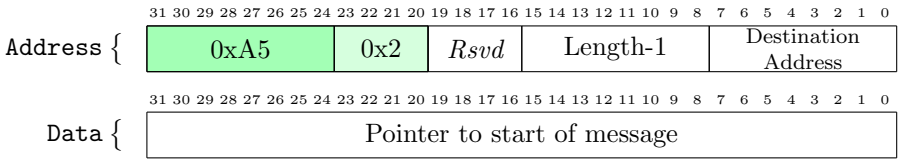
Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
8	Message Destination Address
Data	
32	Data to send on MBus



4.3.2 Arbitrarily Long Write [Write Only]

This transaction requires a buffer to read from and a length. Instead of duplicating memory units and copying a buffer into the layer controller, the layer controller instead DMA's directly from memory. This means the layer controller should have enough local buffer to always stay ahead of the bus controller and have data prepared in case it loses memory bus arbitration.
For the programmer, this means the region pointed to during this transaction is immutable. It is undefined what occurs if a write to the transmission buffer is attempted.

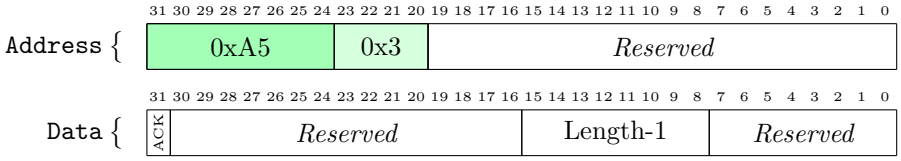
Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
8	Message Destination Address
8	Number of *words* to send (max 256)
Data	
32	Pointer DMA buffer start



4.3.3 Transaction Result [Read Only]

A register that stores the result of the previous transaction request. "Result" is defined as the number of bytes successfully sent on the bus before a reset event occurred and whether the transaction was ACK or NAK'd. Note that the ACK bit will only be set if the entire transaction completed successfully and was ACK'd. The length field is primarily useful for a NAK'd transmission as a *hint* to the programmer of what *likely* went wrong. If a message is ACK'd, the length field must match the length of the request.
The register should block, not returning from the read request until the bus transaction is complete (though see 5.1 for some concerns). A read of this register before a transaction has been request will return with a NAK, the rest of the register contents is undefined.

Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
Data	
1	ACK (1) or NAK (0)
8	Number of *words* sent



4.4 Transaction RX

4.4.1 RX Interrupts

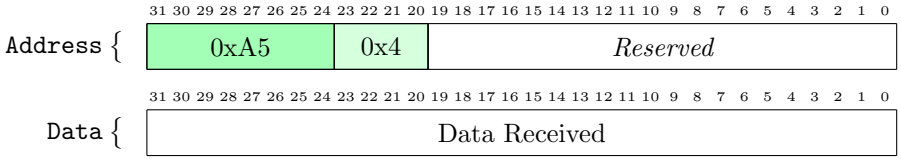
RX Dropped A reception was dropped (bus controller sent reset / did not ACK) by the bus because there was not enough room to store the message. This interrupt may be suppressed by 4.2.1.

RX Received A message has been received. Depending on the expense of interrupts (TODO **??**), this may be a unique interrupt per RX Buffer or a single shared interrupt with a register to read where a message came in. This interrupt may be suppressed by 4.2.1. **Disabling this interrupt will disable message reception.** The bus controller will NAK even single-word messages if this interrupt is disabled.

4.4.2 RX Word (single-word) [Read Only]

Similar to TX, a shorter path to receive a single word message. If another message is received prior to this message being read out, the new message should be NAK'd.
The layer controller may include a small internal queue of received messages. In that case, the RX Received interrupt (4.4.1) should remain asserted until all messages have been read out. This will ensure software cannot disable reception until all messages have been received.

Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
Data	
32	Data received



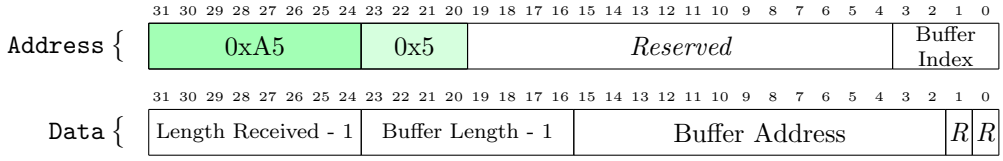
4.4.3 RX Buffer (multi-word) [Read/Write]

Incoming messages need to be stored somewhere, they also may take a non-zero time to copy out of memory, or in many cases actually copying the message may be a completely unnecessary and wasteful operation.

The layer controller has NNNN configurable message buffer targets. These targets hold a memory address and a maximum length in words. The target address must be word-aligned, however the bottom two bits of the address are reserved for possible future use. Software should take care to write in word-aligned addresses to this field and to clear the bottom bits when reading back before use in case they are used by a future implementation. When the layer controller uses a buffer to store a message, it sets the length of the received transaction, which marks that buffer as invalid for future use by the layer controller.

Out of reset, the RX'd message length is non-zero for every target, meaning the layer is incapable of receiving multi-word messages directly out of reset. For programs not designed to receive large messages, this will set the M3 system up to automatically NAK long transactions with no programmer intervention.

Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
2-4	Buffer Target Index
Data	
8	Length of RX'd message
8	Length of buffer (256 *word* max)
16	Buffer memory address



5 ToDo

5.1 AMBA Bus Contention?

When a programmer requests the result of a transaction, it would be desirable to block the CPU until the transaction is actually complete. In practice, this was previously implemented by holding the `H_READY` line of the AMBA bus low until the bus transaction is complete. If the layer controller needs to access memory, however, to read the next word it wishes to send on the bus, then we will possibly be in a dead-lock situation?

Does AMBA have a mechanism to allow others to use the bus while a long-stalling request is active? Or is part of the AMBA specification that accesses should not stall for a long time as they lock up the bus?

One solution could be using the high bit of the register to indicate whether the transaction is finished or not, which would allow the response to return immediately. This would cause repeated polling reads of the status register, which isn't ideal. That could be solved by having a "sending transaction complete" interrupt, which would allow the processor to enter WFI, but that begins to add a lot of complexity to software to send a simple message (that said, such complexity is certainly not unheard of, it's how most radio drivers work, but those are usually much longer transmit events. It would be a little heavy-handed for every local bus transmission I think).

5.2 RX / TX Buffer size?

In the current design, I allow for a maximum TX and RX buffer of 2^8 words (1 kB) of data. While this feels reasonable for a chip with only 3 kB of memory, it already starts to feel somewhat restrictive for a chip with 16 kB (extended version).

How big are the images that the imager layer captures? What is a reasonable maximum buffer size? What was the maximum message length supported by I2C?

6 Document Revision History

- Revision 0.3 (rXXXX) – Xxx X, 2013
  - Add functional unit addresses
  - Change addressing to reflect new prefix-style addressing
  - Update power-gating details
  - Add Wire Controller
  - Add Interrupt Controller
  - Add Channel 3 Information
  - Update layer controller information to match actual design
  - Use the word *Reserved* in the programmer model
- Revision 0.2 (r3194) – Mar 4, 2013
  - Add power-gating information
  - Add `RX_FAIL` signal
  - Solidify Interrupt TODO
  - Add `PRIORITY` signal
- Revision 0.1 (r2847) – Jan 22, 2013
  - Initial revision

A M3 Register/Memory Maps

A.1 M3 Legacy Chips

Questions for People:

I tried to guess at who can answer all the missing pieces:

Yoonmyung

Gyouho

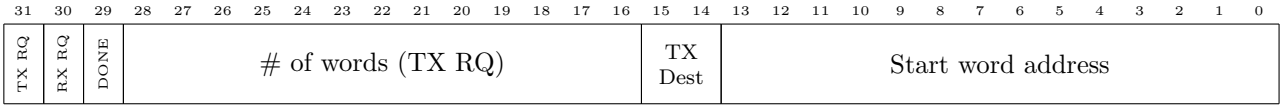
A.1.1 Address Space Usage

	Prefix	Addr	R/W	Data Field	Description
CTR Layer	1000	00XX	W	32-bit Data	Write data to the CPU's message registers MSG_REG0 - MSG_REG3.
	1000	01XX	W	32-bit Data	Write data to the CPU's interrupting message registers INTMSG_REG0 - INTMSG_REG3. An interrupt (IRQs 16–19) will be generated in response to a write of one of these registers.
	1000	1000	W	DMA Data	Data from a DMA transaction. This message must come between DMA Control Messages (start and stop). The DMA transaction may be sent as one long message or a series of messages sent to this functional unit. The length of the sum of DMA data messages must match the length specified by the DMA Control Message that started the transaction.
	1000	1010	W	†DMA Control Message	Control DMA engine. This message sets up DMA transfers. A TX Req instructs this layer to send data, begging at the start word address. A RX Req instructs this layer to receive DMA data and write it to the start word address. Rules surrounding the DONE messages? Sender needs to send it for RX Req, what about TX Req?
	1000	1100	W	?? Data	CHIP_ID Reg
	1000	1101	W	?? Data	DMA_INFO Reg
	1000	1110	W	?? Data	GOC_CTRL Reg
	1000	1111	W	?? Data	PMU_CTRL Reg
	1000	1001	W	?? Data	WUP_CTRL Reg
	1000	1011	W	?? Data	TSTAMP Reg
DSP Layer	1001	000-	R/W	32-bit Data	Status Register more detail?
	1001	001-	R/W	32-bit Data	General purpose register #1, non-interrupting?.
	1001	010-	R/W	32-bit Data	General purpose register #2, non-interrupting?.
	1001	011-	R/W	32-bit Data	General purpose register #3, non-interrupting?.
	1001	1000	W	DMA Data	*<See CTL Layer DMA>*
	1001	1010	W	†DMA Control Message	*<See CTL Layer DMA>*
	1001	110-	R/W	32-bit Data	Interrupting message register (IRQ0)
	1001	111-	R/W	32-bit Data	Interrupting message register (IRQ1)
	1011			*RADIO Layer — Currently Undocumented*	
IT Layer (Imager)	1100	000-	R/W	13-bit Data	The CLKX2REG register. This register does X, can be set during Y, etc?
	1100	001-	R/W	22-bit Data	The IMGREG register. This register does X, can be set during Y, etc?
	1100	010-	R/W	30-bit Data	The SLOWTIMEREG register. This register does X, can be set during Y, etc?

Prefix	Addr	R/W	Data Field	Description
1100	011-	R/W	ResetRequest, 27-bit Data	Bit 28 is the ResetRequest register. The ResetRequest bit is not written, rather forces a reset of (?) the whole layer. If the ResetRequest bit is set, the temperature tracking register is/is not set. Bits 0-27 are the TEMPTRACKREG, which has no documentation.
1100	1000	W	†Prep Take Image	This is all guesswork: This command wakes the actual imaging circuitry, optionally reporting back when the wakeup is complete. The imaging circuitry must be woken independently of the imager layer itself and must be woken before an image can be captured.
1100	1010	W	[7:4] DMA target addr, [3:0] Wakeup delay (/16)	Take image command. Bits 7-4 set the layer ID (short prefix) to send the DMA transaction to. Guess: Bits 3-0 are a delay between receiving the message and triggering the image capture. This value is multiplied by 16 and counts cycles on the local imager clock, which runs at XXX MHz.
1100	1100	W	[7:0] Response Addr	This message requests the temperature as measured by the imager. Bits 7-0 specify the short address that the response message should be sent to. The response is sent immediately // The response is sent after a XXX [second/cycle] delay while a sample is taken.
1100	1110	W	None?	The message requests a TempCount Report. what is it? Is there any data? Where is the message sent?
1101	000-	R/W	†TTM/TEMP Config	Read/Write the TTM, TEMP, and TTMWAKEDLY register configuration. Guess: Temp Report addresses control where TempCount Report messages are sent. TempFreq?, Reset Delay? Wakeup Delay?
1101	0010	W	[7:0] Response Addr	Measure fine-grained oscillations and report. The value reported back means...? The response is sent to the short address specified in bits 7-0.
1101	0100	W	[8] Slow Timer, [7:0] Response Addr	Take timestamp command. This message takes a timestamped picture // reports back the local time?. The slow timer always reports its value. If bit 8 is set, the fast time also reports its value. The response is sent to the short address specified by bits 7-0. The response format is...
1101	0110	W	*<ignored>*	This is a dummy message used to wakeup the imager. The data is ignored, but must be non-zero in length to ensure that the bus controller wakes the layer controller to handle the message.
1101	100!	???	???	Setup TTM1 Timestamp?
1101	101!	???	???	Setup TTM2 Timestamp?
1101	110!	???	†Setup TTMX Message	What is a TTM Message?
1101	111!	???	†Setup TTMX Message	What is a TTM Message?

IT Layer (Timer)

DMA Control Message

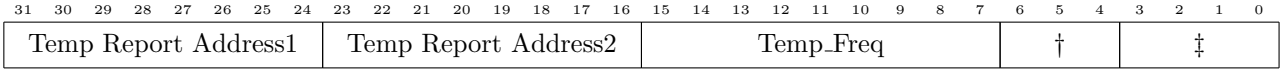


TX Dest— 00: DSP, 01: R-SRAM, 10: RF, 11: Imager

Prep Take Image

`EN`: Set to `1` to enable the transmission of a done message.
`LEN`: Length of the done message. What does this mean?

TTM/TEMP Config



† Temp_RESETDELAY
‡ TTM wake up delay (×16)

Setup TTMX Message