MBus Specification

<mbus-team@umich.edu>

Pat Pannuto <ppannuto@umich.edu>
Yoonmyung Lee <sori@umich.edu>
Ye-Sheng Kuo <samkuo@umich.edu>
ZhiYoong Foo <zhiyoong@umich.edu>
Ben Kempke <bpkempke@umich.edu>
David Blaauw <blaauw@umich.edu>
Prabal Dutta <prabal@umich.edu>

Revision 0.3+ — Working Revision

Overview

MBus is an ultra-low power system bus. The original design was motivated by the Michigan Micro Mote (M3) project. The goal of MBus, however, is to be a general purpose bus for hyper-constrained systems. MBus requires four pins per node, uses purely digital logic, supports arbitrary length transfers, and features a low-latency priority channel and robust acknowledgments. MBus member nodes do not require a local clock and are capable of completely clockless operation. MBus requires one more capable node to act as a bus *mediator* node, whose primary duties are providing the MBus clock and mediating arbitration.

Contents

1 Requirements & Design Considerations

This section attempts to explain the design tradeoffs made in the MBus design.

Define bus PHY, MAC and application logic interface We take a vertical integration approach, controlling more of the complete stack in an effort to obtain every available ounce of energy efficiency within our further design constraints.

Must be fully-synthesizable In an effort to make MBus ubiquitous, it is important that other designers can easily add MBus to their systems/chips. In practice, this means a fully-synthesizable design such that simple, pure Verilog can be given to any individual wishing to implement MBus. This provides added advantages in pre-silicon testing and validation.

Optimize for synthesis simplicity over speed The synthesizability is paramount. As example, standard cell libraries contain both positive and negative edge triggered flip-flops, but a dual-edge flip-flop is not available in many processes. It is important then that no element of MBus design require an implementation that double-clocks flops.

Must be low-power, low-leakage The MBus was designed for the Michigan Micro Mote (M3) project. The system is designed to run off a 0.5 μAh battery. To serve as a feasible communication bus, the energy consumption must be on the order of pW.

This extreme power constraint eliminates traditional analog open-collector style designs as options. Sufficiently low-leakage circuits would have untenable drive strength requirements and/or rise time constraints. The previously stated portability requirements eliminate custom crafted analog elements as an option, restricting MBus design to a purely digital approach.

Support Multi-master operation / bus arbitration / avoid race conditions As every layer is capable of entering an extreme low-power state, any layer must be capable of waking the system. The waking layer must also be capable of communicating (i) who woke the system and (ii) why. While such a system could query (ii) given (i), determining who woke the system requires a complex dedicated controller, or for the waking layer to simply announce it. A multi-master bus greatly simplifies the software design and allows for things such as DMA from an imager chip to a radio chip without necessarily requiring CPU intervention.

Once multi-master is agreed upon, some form of arbitration scheme must be devised, as well as consideration for races as multiple layers attempt to acquire the bus.

Minimize pins

Minimize gates

Limit the number of I/O pads The physical area of the M3 system is extremely constrained. There is only room for four I/O *pads* on each layer of the system (keep in mind that forming a net/bus still requires two *pads* when wirebonding).

Minimize timing uncertainty/latency/jitter

Allow for wakeup that takes time As the system supports extremely low-power sleep, the bus must tolerate / account for a lag in waking layers of the system. In practice this is mostly a focus on the bus layer controller for each layer at first, followed by layer-specific concerns of how much local buffer should be necessary and when/how to wake the proper layer.

Active vs sleep mode; want backward compatibility The bus needs an idle state that consumes very little power. Ideally, each layer not participating in the current transaction also burns very little power.

There should also be some consideration for future-proof design, that is the introduction of a newer protocol that can run on the bus that older controllers can silently ignore.

Ensure that bus reset is possible Stuff happens. It is important that there exists some kind of escape hatch to rescue / reset the bus. For deployed systems with multi-year lifetimes, the ability to reset to a known state from nearly any possible erroneous state becomes critical. Details of reset design are discussed in 1.1 Reset Design.

Support variable-length packet sizes Short packet length limits (e.g. order 4 bytes) lead to unacceptably high protocol overheads for large (order 1 kB+) messages. An upper bound on packet-length leads to possibly complex software-level fragmentation schemes (or worse hardware-level if the bus packet length < message buffer).

A leading length field with a reasonably large upper bound would then be acceptable, although not ideal. The goal is to allow for true arbitrary length packets if possible.

Consider automatic address space management protocol? There is an extensibility / simplicity trade-off. I2C only allowed for 128 addresses, which for an individual bus instance was seen as more than enough (still pretty true with the loading requirements, but repeaters / extenders make that less true). The real issue though is address space collisions between generic components. To build a plug 'n play bus of any arbitrary pieces requires the whole corpus of parts to have non-conflicting addresses. This motivates extensibility.

MBus strikes a balance in its address design, using short, 8 bit addresses in the common case but defining a full, 32 bit address to resolve address space collisions. Addressing details are discussed in 9.10 Addressing.

Allow interposing of the bus to read out / insert data from external source Sniffing pretty much any bus design is easy, but adding devices post-hoc is a harder requirement. There is great motivation for transient connection of devices (programmer, debugger, etc), and also some motivation for the ability to permanently add new devices to an existing bus.

As MBus employs a ring style connection, injecting a new device in an existing bus may not be trivial. Interposition of external devices is not seen as a requirement for MBus to define, however, simply as something to caution system implementers as desirable. For one example of bus injection with minimal pad/pin overhead, consult the M3 Implementation document.

All wires must be uni-directional Bi-Directional analog hardware is complex and not synthesizable.

Granularity Transmissions are required to be modulo 8 bits in length. The motivation for this is largely a function of the arbitrary nature of the last two bits received. It truly arbitrary lengths were allowed, an interruption occurring after 9 bits were transmitted could be any of 0, 1, or 2 bytes long (0 bytes: out-of-band interruption, 1 byte: End of Message where receiver precedes transmitter and thus discards two bits, 2 bytes: End of Message where receiver follows transmitter and thus all 9 bits are valid).

This hypothetical receiver's design is further complicated if the receiving node is (locally) clockless. The bus interface cannot reliably hand off any bytes until it receives the *11th* bit. It cannot reliably learn that it has received two bytes until the Begin Control edge. There are only three bus clocks between that edge and Idle, and only one edge until the receiver would be obligated to ACK or NAK, a very challenging design constraint to occasionally omit up to 7 bits from a message.

Endianness

Byte-Level No specific design consideration here per se. Given that MBus supports messages of an arbitrary length, it simply felt more logical to send bytes from 0...N instead of N...0.

Bit-Level Again an arbitrary decision. The first test implementation elected to use MSB and the decision was made.

Retransmission Hardware retransmission carries risks of accidentally locking up the bus, endlessly retransmitting. While ideas such as a maximum retry count (SMBus) mitigate this, the added complexity and risk of endless hardware retransmission tip the scales in favor of pushing the retransmission decisions to software.

To provide the software layer with more information, we choose to require the indication of the number of bytes actually transmitted. While transient faults could certainly occur, it is reasonable that the software can infer different *probable* causes from the amount of the message that was transmitted.

Flow Control Any form of flow control requires communication *mid-transmission* between the transmitting and receiving node. This could be accomplished by periodically (e.g. every word) switching roles, but this introduces a large amount of complexity. Another solution is to enable some form of clock stretching, but these designs are either analog in nature (e.g. I2C) or prohibitively complex to implement.

As MBus supports arbitrary interruptions, a simpler design is possible. If a receiving node's RX buffer is overrun, it interrupts the bus and indicates a buffer overrun. Not interrupting the bus acts as an *implicit* flow control. In practice, transmitting nodes should consider sending some form of *ping* message to a destination node to validate its presence before sending a large transmission to minimize waste if the destination node is not actually present. MBus is not designed for a changing topology, as such discovery of this nature need only be performed once and the amortized cost is considered negligible.

Acknowledgment granularity As MBus supports messages of arbitrary length, a natural question arises for the granularity of acknowledgments. At one extreme, designs such as I2C elect to acknowledge every byte. MBus takes the opposite approach, providing only a single acknowledgment at the end of a transmission in an all-or-nothing fashion. The rationale for this decision again returns to simplicity and efficiency in design. While a receiver-driven acknowledgment cycle could easily be added every nth bit, the relative merits of this are not great. Assuming a receiver exists on the bus, by not electing to Interrupt transmission, a receiver implicitly ACKs every bit sent. A receiver that has occasion to NAK a transmission may do so at any time during transmission. The obvious weakness of the implicit ACK is an absent receiver, however MBus expects a static topology and thus this is not considered an important consideration.

Streaming Some bus designs incorporate a "streaming" mode, which allows for a series of physical bus transactions to be considered one contiguous message at the receiving layer. As MBus allows for arbitrary length messages, such a feature becomes largely unnecessary, and MBus defines no streaming primitive.

Avoid ratioed-logic, avoid timing constraints

Idle State Should Be High For designs that aggressively power-gate components, it is easier to build designs that pull low in minimal power modes than pull high. As a consequence, the Idle state in MBus elects to leave all lines high.

1.1 Reset Design

The reset mechanism is the MBus escape hatch. Its reliability is critical such that no matter what state MBus nodes are in, they can all be reliably forced back to a known state. Special care must be taken that no assumptions are made about the state of any node in the system. As example, in normal operation only one node should ever not be forwarding. The reset mechanism, however, must correct for more broken cases where any N nodes are not forwarding and return all nodes to a common and known state.

1.1.1 The Reset Detector

MBus's reset detector is advantageous in that it is completely isolated from the MBus state machine. To reset MBus, the Interrupt sequence is entered. This entry saturates, allowing nodes to be 'endlessly-interrupted' until all of the nodes have been interrupted. A misaligned state machine, or a timing glitch, or other failures related to normal operation cannot affect the interrupt entry detection, as the detector is an isolated, dedicated circuit. The exact requirements for the Interrupt detector are discussed in 3.4 Interjection.

1.1.2 Hung Nodes

We define "hung" as an erroneous state from which a node would never depart without external intervention. We are not concerned with how a node got into an erroneous state—a pathological series of SEUs, whatever—the focus is on how a node excises itself.

Busy During Bus Idle In this state, the MBus is Idle, but a node believes there is still a transmission taking place and will wait forever for a new clock edge, constantly reporting the MBus as busy.

There is a small window where it is possible to enter this state as a node must error during one of the bits in the control sequence. A node will be stuck in this state until another transmission occurs. The behavior of an erroneous node is not well-defined.

> *Aside:* A node with its own local sense of time could include a mechanism that bounds the period of time without a pulse on the clock or data lines, but such a mechanism is outside the scope of this document. The MBus neither prohibits nor requires clock stretching. A minimum clock period is specified as a function of MBus topology, but a maximum is not. If building a design with such an escape mechanism then, designers must be conscious of the implementation details of its master node.

Endless TX If a transmitter enters an error state and never interrupts the bus, other nodes will not be able to distinguish an endless transmission from a transmission with a long string of `0`'s (or `1`'s). MBus relies on the controller to detect the error. While MBus does allow for arbitrary message lengths, a master node must define a maximum message length for a particular MBus instantiation[1]. The master node counts the number of bits received and can forcefully reset the bus if necessary. Address bits count towards this limit. Interrupt entry edges do not count towards this total as the rising clock edges will not be seen by the master node's `CLK_IN`. The intention of the limit is not to be used ever during correct MBus function, and thus should be set well above the maximum expected message size. The absolute minimum limit for a conforming MBus is 1 kb ($[1024 - addr\_len]$ data bits).

1.2 Power Gated Nodes

As MBus is designed for extremely low-power systems, it is reasonable to address the needs for the coordinated wake-up of power gated nodes. A power gated device requires a series of well-organized steps to awake in a known state. This waking is much like a standard chip power-on-reset sequence, only MBus design seeks to avoid imposing custom delay elements on the construction of a node's bus controller.

The following sequences are designed for nodes that are possibly receiving the transmission. Waking the transmitting node is left to alternative means by the implementation, with the tacit assumption that if a node has something it wishes to transmit, it is probably already awake[2].

The generalized wake-up sequence for power-gated devices requires up to four signals:

- Power on (remove power-gating)
- Warm up local oscillators
- De-assert Reset

[1] It is suggested that this value be programmable, for maximum flexibility.

[2] Though, a design that wished to exploit these pulses could take advantage of the spurious wakeup rules described in 9.13. Allow the first transmission to fail and use the available pulses to wake itself. This design would still require some care, as the node would have to begin forwarding its data line again before the arbitration pulse. Using the falling edge of the clock line as the transition from pulling data low to forwarding would be sufficient to achieve this goal.

- Remove isolation

Power gated MBus member nodes are able to use the bus's clock line to provide all four of these edges. The first four positive clock edges of a MBus transmission are: arbitration, priority drive, priority latch, and begin transmission, none of which require receiver intervention. The fifth clock edge latches the first bit of the transmitted address. The implication is that a node using MBus clock edges to wake from power-gating must reset into a state that is prepared to receive the first address bit.

Sleeping a node is a simpler process, requiring only two signals:

- Isolate device
- Power off (power-gate)

By the time a node is sending (or forwarding) the second control bit, it has decided whether it will be putting itself to sleep or not. It a node intends to sleep, it can use the clock pulse that latches the second control bit and the pulse that formally transitions the bus to idle as the two required edges.

1.3 Design Caveats

Here we attempt to draw attention to some of the systems-level issues that may be presented to persons using MBus.

1.3.1 Broadcast Message "Acknowledgment"

For point-to-point messages, MBus acknowledgments provide a strong guarantee to higher layers:

> *The message was received in full or the message was not received at all.*

For broadcast messages, the MBus semantics are weaker:

> *The message was received in full by at least one node or the message was not received at all by any node.*

Systems cannot rely on a "successfully" broadcast transmission as *guaranteeing* that every node has received the message. If an absolute guarantee is required, systems must build one atop point-to-point messages.

1.3.2 Interruptions: Arbitrary Length Messages and Forward Progress

MBus design provides a powerful tool: the ability for any node to interrupt any message at any point. Such a mechanism invites risk of live-lock on the bus, however.

As a first example, consider two nodes that simultaneously wish to send an interrupt-worthy high-priority message. One node will win arbitration and begin transmitting, only to be immediately interrupted by the other. The two nodes will continuously interrupt one another in an effort to send their very high-priority message, and neither will ever succeed.

Another form of live-lock stems from periodic low-latency messages. Consider a system with a timer chip that needs to generate periodic 100 ms pulses that another chip will need to respond to in a bounded latency. The responsiveness requirement will place an interrupt-class priority message on the bus every 100 ms. On a 400 kHz instantiation, if another node (say a camera node) attempts a bulk transfer (an image) it will only be able to send approximately 40 kilo*bits* before being interrupted. A picture over 5 kB will never send.

As the only physical bus available for a complex system, MBus is required to support both timing-critical messages and large data transfers. To provide sufficient flexibility to efficiently handle both types of messages while maintaining very simple node architectures (e.g. no suspending transfers), MBus pushes additional considerations to the systems design.

MBus provides both some requirements and some guidelines to help system designers:

Requirements These are elements of the MBus specifications designed to help systems with bus contention issues.

- The first 32 bits of a message may not be interrupted

- There is no means in MBus arbitration to ensure all nodes can distinguish priority and regular messages[3]. As such, MBus guarantees that at a minimum 32 bits of data may be transmitted without interruption (**?? ??**).

Guidelines These guidelines operate under the assumption that a priority message is of sufficiently high priority that it would warrant interrupting an active transmission. Without preemption, the livelock scenarios are averted, however the issues of starvation remain.

- Priority Message Length
  - Priority messages are permitted to be any length, *however*, only the first 32 bits—even for priority messages—are protected from interruption. A design that sends two interrupt-worthy high-priority messages longer than 32 bits *will* livelock.
  - MBus strongly *recommends* that all priority messages be restricted to 32 bits. With great care and caution designers may violate this constraint.
- Priority Message Frequency
  - As priority messages also have a topology-dependent component, MBus advises that after sending a priority message a node back off from sending *any* message (regular or high priority) for at least 80[4] bit-times per MBus node.
  - This is not a formal MBus requirement as it is a recognized design constraint that not all member nodes are required to have any sense of time. Furthermore, it is believable that such time-less nodes may be sensors that require a low-latency response. It is not the intention of a bus design to define how such an issue is dealt with. It is, however, then the obligation of the designer of such a sensor to carefully consider how to be a good MBus citizen and how to avoid saturating the system's only communication bus.
- Normal Message Length
  - On a system with interrupts
    - The first concern must be successful forward progress in the face of interrupts. Long messages must be packetized (most pathologically down to 32 bit windows, less headers). Future MBus designs consider more amenable solutions to this drawback, see: **?? ??**.
    - In practice, consider the shortest interval of interrupting messages your system may encounter. Multiply this window by the bit time and halve the resulting value to get an approximate maximum packet size.
  - On a system without interrupts (or with infrequent ones)
    - Long messages cause starvation. You must consider the acceptable average latency and use this coupled with the bus speed to define a reasonable maximum message size.
    - Frequent short messages can cause starvation. While nodes are not required to have a sense of time, those that do should back off for a window of time between messages. Nodes without a sense of time should be designed with avoiding flooding in mind.

[3] e.g. in **??**, Node 2 would have seen the exact same signals if Node 3 had not participated at all.

[4] Arbitration (4) + t_{long} (2) + address (8 or 32) + data (32) + interrupt (6) + control (2) + idle (1) = 79.

2 Node Design

The MBus defines two *physical* types of nodes: member nodes and a mediator node. An instantiation of MBus must have one and only one mediator node and must have at least one member node (N≥1). The maximum number of member nodes is a function of clock speed[5].

During a transmission, MBus defines three *logical* types of nodes: a transmitting node, a receiving node, and forwarding nodes. During a transmission, there must be exactly one transmitting and one receiving node. Any number (N≥0) of forwarding nodes are permitted.

2.1 Physical Design

From a package perspective, the physical design of a member and mediator node is the same, each must expose a `DIN`, `DOUT`, `CLKIN` and `CLKOUT` pad.

In addition, one (or more?) chip(s) on the bus can add two additional pads to act as a "splitter" node, to allow the interjection of new devices on the bus. This part of MBus is not yet well-defined.

2.1.1 Member Nodes

A member node requires 4 signals:

- `DIN` – Data In
- `DOUT` – Data Out
- `CLKIN` – Clock In
- `CLKOUT` – Clock Out

Most of the time, a member node is in the FORWARDING state. While forwarding, a member node must amplify and forward signals from the `DIN` pin to the `DOUT` pin and from the `CLKIN` pin to the `CLKOUT` pin. Designs should attempt to minimize latency between these pins. The maximum propagation latency[6] permitted is 10 ns. MBus defines a maximum load capacitance of XXX pF for the `DIN` pin and of XXX pF for the `CLKIN` pin to enable inter-operability of generalized components[7].

2.1.2 Mediator Node

A mediator node requires 4 signals:

- `DIN` – Data In
- `DOUT` – Data Out
- `CLKIN` – Clock In
- `CLKOUT` – Clock Out

While forwarding, the mediator node is subject to the same propagation latency constraints (10 ns) as member nodes.

2.1.3 Bus Connections

- `DIN`, `DOUT`
  - The data pins shall be connected in a round-robin fashion, the `DOUT` of one chip connected to the `DIN` of the next.
  - The connection of data lines must form a loop when connected correctly (e.g. **??**).
  - There are no requirements for the placement of nodes in the data loop, but the ordering will have an impact on bus arbitration. See **??** for more details.

[5] The maximum chip-to-chip latency is defined as 10 ns. Assuming a TX node has similar delay to generate the next bit and clock delay to nodes is 0, then the minimum cycle time is N nodes * 20 ns. MBus designers are obligated to consider all possibly delays and thus define a maximum feasible clock speed, however, for most designs the maximum clock speed theoretically possible will largely exceed the plausible speed for the given power budget.

[6] The amount of time taken to propagate a change in the output of the *previous* link in the data loop to the output of the next link in the data loop. This time includes all of the internal forwarding logic from `DIN` to `DOUT` or `CLKIN` to `CLKOUT`, as well as any pin/wire capacitance that must be overcome to drive the next input gate.

[7] However, the only strict requirement is the 10 ns propagation delay, thus careful designers may violate this requirement if necessary.

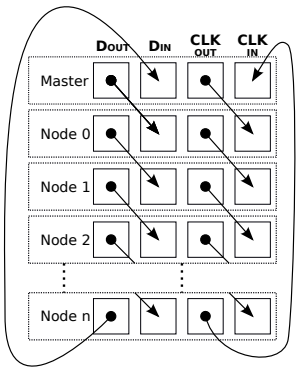


Figure 1: MBus Physical Topology. High-level picture of MBus physical design. Member nodes and a mediator node are connected in a loop, with data and clock lines forming independent rings.

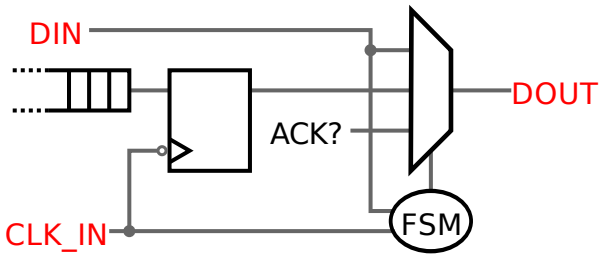


Figure 2: Logical Model. The Finite State Machine selects between the three modes a node can be in. Top: *forwarding*, Mid: *transmitting*, Bot: *acknowledging*. This model omits some of the subtleties of arbitration, see 2.2.2 Transmitting.ARBITRATE for details.

- CLK
  - The clock pins shall be connected in a round-robin fashion. The CLKOUT of one chip connected to the CLKIN of the next.
  - The connection of clock lines must form a loop when connected correctly (e.g. **??**).
  - The connection of clock lines must match that of data lines. That is, if a node N_a connects its DOUT to the DIN of node N_b, the CLKOUT of N_a must be connected to the DIN of N_b.

  A MBus must have a minimum of two nodes (one mediator node and one member node). Single chips that are not connected to a bus should tie CLKIN and DIN high and leave CLKOUT and DOUT floating.

2.1.4 Injection

Injection is the ability for an arbitrary additional node to temporarily (or permanently) interpose itself into a MBus instantiation. Two examples are a system programmer or a debugger.

The exact method of injection is not defined by MBus. It may be as simple as exposing a pair of out/in pins that are normally jumpered, some packages may not have such luxury. Injection is mentioned here as it is an exceptionally useful tool for system bring-up and development. The means for performing injection should be considered as a MBus instantiation is designed.

2.2 Logical Design

There are three *logical* types of MBus nodes: transmitting, receiving, and forwarding. MBus can be considered multi-master, any node is capable of transmitting to any other node. The mediator node adopts these same three personalities, but its behavior differs slightly during arbitration (3.2).

2.2.1 Forwarding

Forwarding is the most common state for all MBus nodes. Observe in Figure 2 the very simple, short logic path from DIN to DOUT. Nodes are obligated to forward data in less than 10 ns.

Forwarding.Idle This is the rest/idle state for MBus nodes. In this state member nodes may be completely power-gated and asleep. The only obligation is that DIN is forwarded to DOUT and CLKIN is forwarded to CLKOUT.

Mediator Node Exception: In IDLE, the mediator node does not forward DIN to DOUT.

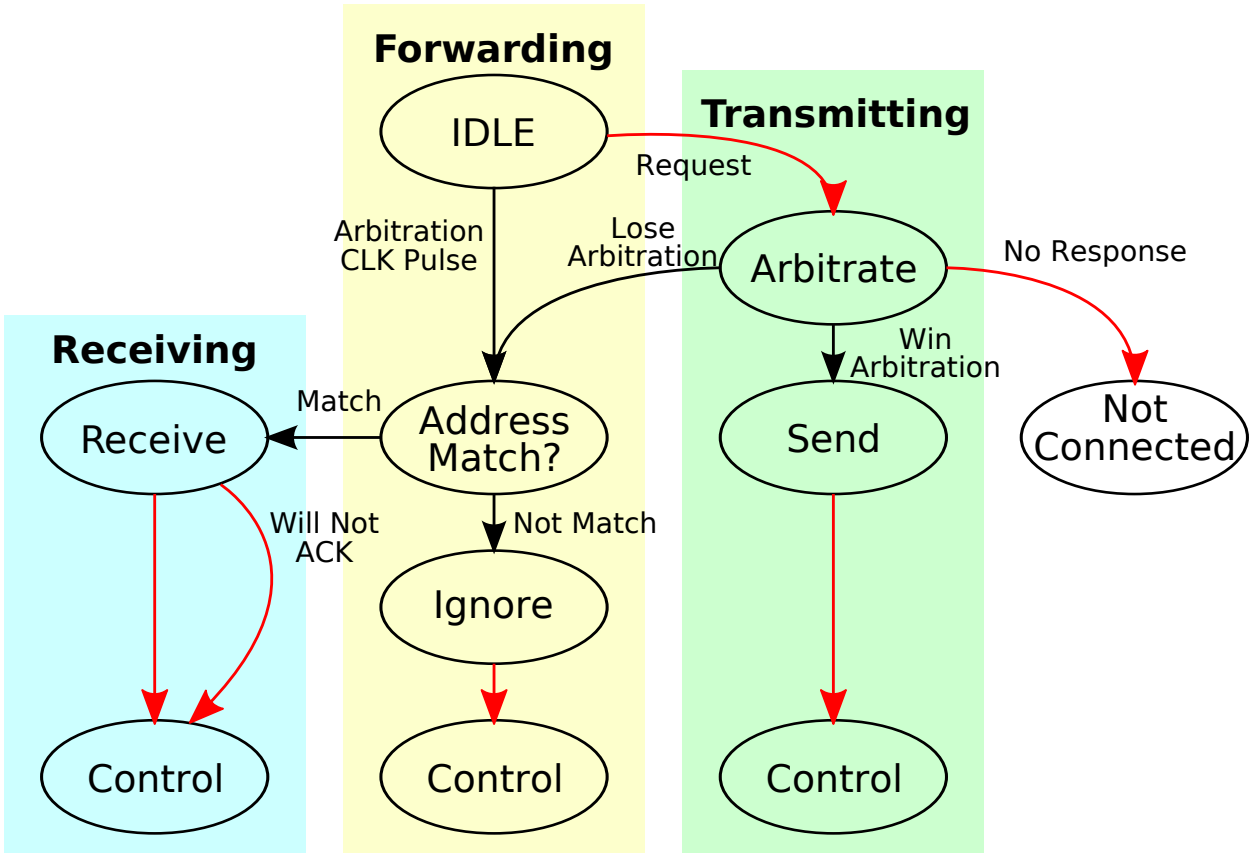


Figure 3: FSM describing the high-level logical behavior of MBus nodes. Black arrows indicate transitions that occur on bus clock edges. Not shown are implicit arrows from any state to Forwarding.CONTROL. From any CONTROL state, nodes progress to IDLE.

Forwarding.Address_Match At the start of a new transmission, a forwarding node should monitor the DIN line to see if it is the target for this transmission. If a node matches its address, it promotes itself from forwarding to receiving. After the first mis-matched bit a forwarding node transitions to the IGNORE state for the rest of the transaction.

Forwarding.Ignore In IGNORE nodes simply forward data. Nodes remain in this state until an interjection. Power-conscious designs may safely power gate everything except interjection detector circuitry while in IGNORE.

2.2.2 Transmitting

Transmitting.ARBITRATE A node initiates a transmission by pulling its DOUT line low. A node may only attempt to initiate a transmission while the bus is idle. Care must be taken in detecting the bus idle state when requesting to transmit. In particular, a node requesting to transmit must ensure that the CLK line is still high, it is not sufficient to rely on the local state machine still being in the IDLE state[8].

The ARBITRATE state is left when the CLK line is pulled high by the mediator node. If a member node's DIN is high on the rising clock edge, it has won arbitration. A node that loses arbitration should begin

[8]To envision the case defended against here, picture a tall stack of nodes, where the bottom node requests the bus. The mediator node pulls CLK low in response. Shortly before the mediator node pulls CLK high, a node at the top of the stack (still in the IDLE state) elects to transmit. There is not enough time for his DOUT to propagate to the bottom node, however, thus when CLK goes high, both the bottom and top nodes believe they have won the arbitration. Designers must select a sufficiently long period t_{long} to ensure all signals are stable. Twice the maximum possible propagation delay (delay for falling CLK to furthest node + furthest node's DOUT back to closest node) of the system should be sufficient.

listening to see if it is the destination node. If the CLK line never goes high—where never is defined as four times the minimum clock speed of MBus[9]—the node should consider itself as disconnected.

Details of priority arbitration are omitted here for simplicity. See 3.2 Arbitration for details.

Mediator Node Exception: The mediator node always wins arbitration. Note that when this occurs, the mediator node's DIN will be low.

Mediator Node Exception: If a mediator node pulls its DOUT line low and its DIN line never goes low, it should be considered NOT_CONNECTED.

Transmitting.SEND During SEND, a transmitting node pushes bits onto the bus as described in 3.3 Message Transmission.

A transmitting node completes its transmission by interjecting (3.4 Interjection) the bus and indicating the transmission is complete.

Transmitting.CONTROL As a transmitter, a node is responsible for the first control bit. This bit should be set by a transmitter to indicate that the complete message has been sent. The transmitter must listen to the subsequent control bits to establish if the message was acknowledged and if the targeted layer is sending a response.

2.2.3 Receiving

Receiving.RECEIVE A receiving node is also obligated to forward data along the bus. In the sketch shown in **??**, during the receiving process, a receiving node remains in pass-thru mode until an interjection.

Receiving.CONTROL A receiver must acknowledge the successful receipt of a transmission. If a receiver wishes to NAK a transmission, it simply does nothing during the ACK/NAK control bit.

A receiver may also possibly enter control by electing to interject the bus itself. A receiver will interject a transmission to indicate that its RX buffer has been overrun and the current transmission must be aborted.

2.2.4 Exception States

Exception.NOT_CONNECTED A robust implementation should include detection of some kind for an attempt to utilize the bus when the node is not actually connected to a bus (so that it may report failure). After a node pulls its DOUT low there is a maximum possible t_{long} of TODO before a mediator node must pull CLK low in response. If CLK is not pulled low, the node should consider itself disconnected and report failure to send as appropriate.

[9] TODO: TBD

3 Bus Design

During normal operation, MBus remains in Bus Idle (3.1). A transmission begins with Arbitration (3.2), then Message Transmission (3.3). The transmitter then Interjects (3.4) the bus and indicates the complete message was sent. During the Control (3.5) period acknowledgment is negotiated. After Control, MBus returns to Idle or may optionally send a response message.

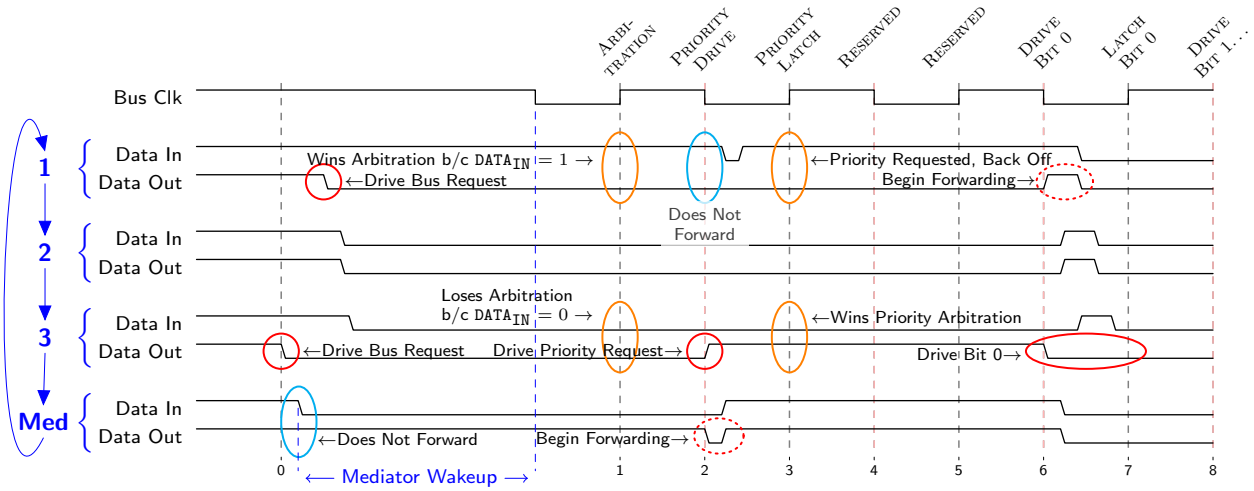


Figure 4: MBus Arbitration. To begin a transaction, one or more nodes pull down on DOUT. Here we show node 1 and node 3 requesting the bus at nearly the same time (node 1 shortly after node 3). Node 1 initially wins arbitration, but node 3 uses the priority arbitration cycle to claim the bus. The propagation delay of the data line between nodes is exaggerated to show the shoot-through nature of MBus. Momentary glitches caused by nodes transitioning from driving to forwarding are resolved before the next rising clock edge.

3.1 Bus Idle

In MBus Bus Idle, all lines (clock and data) are high. All member nodes are in forwarding state and the mediator node is waiting to begin arbitration.

3.2 Arbitration

To begin arbitration, the bus must be in idle state.

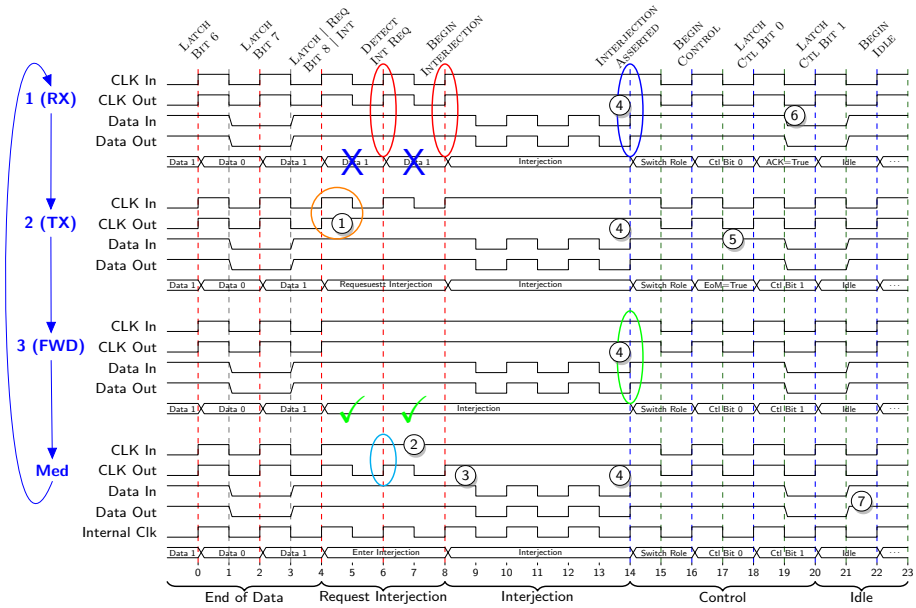
To request to transmit on the bus, a node should pull its DOUT line low. All member nodes remain in forwarding state during arbitration, thus when their DIN line goes low, they are obligated to pull their DOUT line low. The mediator node, however, does **NOT** pull its DOUT line low in response to its DIN line going low. The mediator node only pulls its DOUT line low when it wishes to transmit on the bus. When the mediator node's DIN line goes low, it will pull the CLK line low and hold it low for some period t_{long}. During IDLE (and thus arbitration), member nodes must forward the clock signal.

By the end of the period t_{long}, the effect of the arbitration is achieved. A member node is in one of three possible states:

1. Clock low, DIN high, DOUT high – Lost arbitration (didn't participate)
2. Clock low, DIN high, DOUT low – Won arbitration
3. Clock low, DIN low, DOUT low – Lost arbitration (either lost, or didn't participate)

The rising edge of clock that follows t_{long} commits the results of arbitration and all participating member nodes advance their state machines appropriately.

If the mediator node wishes to transmit on the bus, all member nodes will be in the third state. Note this arbitration protocol introduces a *topology-dependent priority*. Firstly, the mediator node has a greater priority than any member node as its DOUT will propagate around the entire data loop. The priority of the member nodes is inversely related to their proximity to the mediator node in the data loop. That is, the



(a) Interjection timing with successful TX and ACK.

1. After the transmitter sends all of its data it requests interjection by not forwarding `CLK`.
2. The mediator detects that a node has stopped forwarding `CLK`.
3. The mediator stops toggling `CLK` and begins toggling `DATA` – the interjection sequence.
4. After interjection, the mediator begins clocking again. Node 1 discards the extra bits because they are not byte-aligned.
5. The transmitter signals a complete message by driving Control Bit 0 high.
6. The receiver ACK's the message by driving Control Bit 1 low.
7. After control, the mediator stops forwarding `DATA`, driving it high, and returning the bus to idle.

(b) Interjection Events.

Figure 5: MBus Interjection and Control. The MBus interjection sequence provides a reliable in-band reset signal. Any node may request that the mediator interject the bus by holding $\texttt{CLK}_{\texttt{OUT}}$ high. The mediator detects this and generates an interjection by toggling `DATA` while holding `CLK` high. An interjection is always followed by a two-cycle control sequence that defines why the interjection occurred.

furthest node from the mediator node, the node whose `DIN` is connected to the mediator node's `DOUT`, has the highest priority of member nodes. The closest node to the mediator node, the node whose `DOUT` is connected to the mediator node's `DIN`, has the lowest priority.

At the end of t_{long}, the mediator node drives the clock high. The normal arbitration phase ends on this rising edge. The next two clock edges allow for a high-priority arbitration. As MBus priority is topology-dependent, we provide a priority arbitration cycle to allow a low priority member node to preempt transmissions. This priority mechanism is still topology-dependent, that is, if Node 1 in **??** had also attempted to send a priority message, it would have won that arbitration as well. The priority arbitration cycle is two clock edges (one clock cycle), the falling edge after arbitration is for nodes to drive their priority requests onto the bus and the rising edge is used to latch the results. During priority arbitration, the node that won the original arbitration **must not** forward its `DIN` to `DOUT`. At the end of priority arbitration, if the original winner did not request a priority message, the node must sample its `DIN` line. If the line is still low there was no priority message, if the `DIN` line has gone high, however, the node has lost priority arbitration and must back off to allow the priority message requester to transmit.

Whichever node ultimately won arbitration transitions from a forwarding node to a transmitting node. Nodes that lost arbitration revert to Forwarding.ADDRESS_MATCH.

3.3 Message Transmission

The falling edge after arbitration is defined as "Begin Transmission". On this edge, the transmitting node should switch its `DOUT` mux from forwarding `DIN` to the transmit FIFO. Data is expected to be valid and stable during every subsequent rising clock edge.

The first sequence of bits pushed onto MBus are the *address* bits. All nodes on a MBus should listen until their address:

- Matches: Node promotes itself from Forwarding.ADDRESS_MATCH to Receiving.RECEIVE.
- Does Not Match: Node transitions from Forwarding.ADDRESS_MATCH to Forwarding.IGNORE.

There is no protocol-level delineation between address and data bits. The transmitting node sends address+data as a continuous stream of bits (for details on MBus addressing, see 9.10 Addressing). Once a transmitter has sent the complete transmission it interjects the bus.

Legal transmissions on MBus must be byte-aligned (9.1 Granularity). The requirement allows receiving nodes such as node 1 and node 3 in **??** to disambiguate the significance of the last two bits received. A legal transmission will end cleanly on a byte boundary if the node topologically follows the transmitter (e.g. node 3) or will end on a byte+2-bit boundary if the node topologically precedes the transmitter (e.g. node 1).

3.4 Interjection

During normal operation, interjection is only permitted after the first 32 bits of data have been transmitted (9.12 Interjection Rules). The interjection mechanism, however, is also the MBus reset mechanism, and as such member node interjection detectors **must** always be active whenever a node is not Idle.

A MBus interjection is defined as a series of pulses on the data line while the clock line remains high. After N pulses where $N \geq 3$, a node enters interjection[10]. Edges on the data line while the clock is low are ignored. This permits potentially racy changes of `DOUT` drivers to safely change on the falling edge of the clock without potentially introducing spurious interjections.

The first rising clock edge after interjection is defined as "Begin Control". The next two clock edges define the control bits.

3.4.1 Nesting Interjections

In normal operation, interjections are not permitted to "nest", that is, the bus may not be interjected while the control bits are being sent. If an interjection sequence occurs during an existing interjections (which may only occur if the bus is being "rescued" from some erroneous state), the new interjection **must** present control bits `00`.

3.5 Control Bits

The two bits after an interjection are defined as Control Bits. **??** is a flowchart indicating the semantic meaning of the control bits and the resulting state. The first control bit is unconditionally driven by the interjecting node. If a transmitting node has interjected to indicate the end of transmission, it will put a `1` on the bus for the first control bit. In all other cases the interjector must drive `0` for the first control bit.

If the first control bit was `1` then the addressed receiver is responsible for driving the second control bit. If the transmission was sent to the broadcast address (6.1), all nodes—excluding the transmitter which forwards—should drive the second control bit low to acknowledge. The semantic provided to the transmitter of a broadcast message then is either (1) no nodes received the message or (0) *at least* one node received the message.

If instead the first control bit was `0` then the interjector is responsible for driving the second control bit. If the interjector is the transmitter or receiver **and** the issue is related to the current transmission (e.g. receiver buffer overflow) then the second bit should be set high. If the purpose of the interjection is unrelated to the current transmission (e.g. an external, high-priority, time-critical message) then the second bit should be set low.

Unless the interjection carries a specific meaning as outlined in this section, the 00: GENERAL ERROR state should be used for general or unclassified interjections as it carriers the least semantic meaning.

[10] While two pulses is sufficient to distinguish an interjection from normal data transmission, three pulses provides protection against spurious entry to interjection from a glitch on data lines.

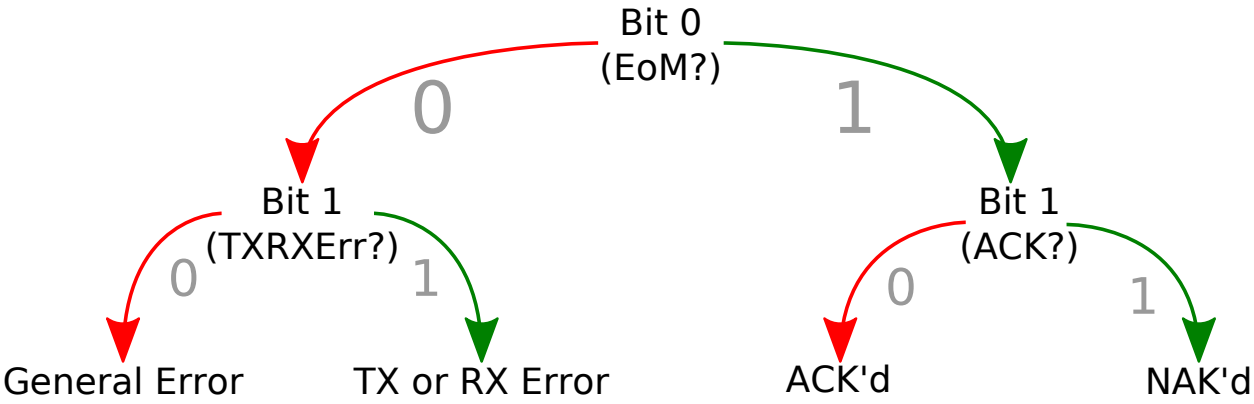


Figure 6: Control Bits. After an interjection, two control bits follow. The first bit is always set by the interjector and indicates whether the interjection was to signal the end of message (EoM). If the EoM bit is high, the receiving node is responsible for driving the second bit to acknowledge the message. If the EoM bit is low, the interjector is responsible for driving the second bit. The TX/RX Error (TRE) bit is set when a transmitting or receiving node encounters an error.

3.6 Return to Idle

After latching the final Control Bit (time 20 in **??**), one final edge (time 22) is generated to formally enter IDLE. If, however, the data line is low at this edge, then it shall be considered the start of a new arbitration cycle instead. The mediator node will pull the clock low again in response and begin waiting for t_{long}.

By providing this edge, MBus enables member nodes with absolutely no sense of time the ability to receive, react to, and respond to messages (albeit with tight timing constraints).

4 Power Design

The purpose of MBus is to support very low power operation. As such, it is expected that systems leveraging MBus may need to support power-gating all or part of the system. In this section, we discuss the requirements to support power-gated systems, how such systems integrate with MBus, and how power-oblivious chips can seamlessly interact with hyper power-conscious chips, promoting interoperability.

4.1 A Brief Background on Power-Gating

In the low-power design space, a simple and important concept is the ability to power-gate, or selectively disable, portions of a system that would otherwise be idle. By power-gating—removing power from that section of a chip—, the power consumption of idle silicon goes to zero.

Several challenges with power-gating include: how to preserve state during idle windows, how to connect power-gated modules to other powered or power-gated modules, and how to wake and sleep power-gated modules deterministically. This document does not seek to address all these issues, rather to demonstrate how MBus can help system designers and the signals that are "safe" to utilize. For a more detailed reference power-gated design with MBus, consult the *MBus M3 Implementation Specification*.

In general, sleeping and waking power-gated modules requires a series of events to occur. In particular, the following signals define common power control design:

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

For the purposes of this document, each MBus node has two modules: (i) The block that interfaces with the bus itself—we define this as the `Bus Controller`—, and (ii) the block that comprises the rest of the node—we define this as the `Layer`. With MBus, a completely power-gated node can seamlessly awaken its `Bus Controller` with no special assistance from the sending node or the mediator node. A `Bus Controller` can filter addresses, only waking the `Layer` for a message destined for that node.

Additionally, a power-gated node with a simple always-on low power interjection generator can exploit MBus features to generate the required edges with no specialization or externally synchronized knowledge of chip status. Finally, devices can reliably detect a shutdown message sent on the bus and use remaining control edges to shut down both the `Layer` and the `Bus Controller`.

4.2 Waking the `Bus Controller`

Referring to edges from **??**, edges 1, 2, 3, and 4 provide the required signals. Mapping power edges to MBus protocol edges:

Arbitration	→	POWER_ON
Priority Drive	→	RELEASE_CLK
Priority Latch	→	RELEASE_ISO
Drive Bit 0	→	RELEASE_RST

In practice, most `Bus Controller` implementations will not require the RELEASE_CLK signal as the MBus clock is (by definition) sufficient for all bus operations, however it is included in considerations for designs that may require it. A `Bus Controller` that is awoken using MBus edges will find its first rising clock edge to be Latch Bit 0, the MSB of the address, and should design state machines appropriately.

4.2.1 Handling an Interjection During Wakeup

By specification, interjection is not permitted during arbitration. If an interjection occurred, an ignorant `Bus Controller` would hang, unable to make forward progress as the remaining edges would have been driven from clocking the control bits, causing the `Bus Controller` to interpret either Latch Control Bit 0

or Latch Control Bit 1 as the MSB of the destination address. After one or two more cycles, the bus would become idle while the local `Bus Controller` waits indefinitely for more bits. This condition would eventually resolve itself if any other node elected to send a message but could not be resolved by any other means.[11]

As the `Bus Controller` was previously powered down, powering it down again before releasing isolation is by definition a null operation. As isolation is released on the final falling edge of arbitration and an interjection may only occur while the clock is high, a wake-up that is not interjected is safe and canceling (re-power-gating) a wake-up that is interjected is safe. The challenge is then that the sleep controller module that generates the power control edges must also be capable of detecting an interjection. Whether this level of robustness is required is left as an implementation decision.

4.3 Waking the Layer

If the `Bus Controller`'s address matches the destination address, it must wake whatever it is attached next up the chain, this means the clockless `Bus Controller` module must harvest clock edges from MBus to generate the power control signals.

One design point explicitly required by MBus is the acknowledgment of zero-length messages. Depending on application, a node may not require awakening for a zero-length message. Due to the nature of the MBus interjection procedure, however, as many as two bits may be received that will be discarded (**??**). A `Bus Controller` design that attempts to minimize wakeups should therefore not begin the wakeup process until latching the *3rd* data bit.

4.3.1 Handling an Interjection During Wakeup

As the control bits provide ample edges, designers have more options for handling an interjection during wakeup. In particular, the same argument regarding wakeup cancellation and arbitration from 4.2.1 applies: if isolation has not been removed, the node may simply be re-power-gated without issue.

A possibly simpler implementation can unconditionally complete the wakeup sequence while indicating that a transaction was started, but failed.

Ultimately, the important consideration is to draw attention to the fact that an interjection *may* occur during the `Bus Controller`'s issuing of wakeup signals (at any point) and a robust `Bus Controller` implementation must consider and handle the cases surrounding interjection during `Layer` wakeup.

4.4 Waking via Induced Glitch

Section 9.13 Failed Arbitration (Spurious Wakeup) defines mediator node behavior in response to a glitch on the data line causing a spurious wakeup. By deliberately inducing such a glitch, a power-gated member node can generate enough pulses to wake itself up. Figure 7 shows an example waveform of an induced glitch, annotated with power control signals.

Details on how the `Bus Controller` disambiguates between an RX-induced wakeup and an interrupt requested wakeup and other implementation issues are outside the scope of this document, but persons developing a power-gated system are highly encouraged to read the *MBus M3 Implementation Specification* as an example of how to design a power-gated system.

4.5 Sleeping the `Bus Controller`, `Layer`

MBus edges can also be harvested to return both the `Layer` and the `Bus Controller` to sleep mode. Figure 8 demonstrates how the control edges following the End of Message bit may be used to put both the node and the `Bus Controller` to sleep.

Figure 8 delays the shutdown procedure until the transmitter asserts End of Message, indicating the shutdown message was not in error. In addition, Figure 8 has the advantage that a node transmitting a shutdown request can itself shut down with no further intervention or knowledge by the mediator node.

[11] Excepting things such as a local timeout and an external reset, but such a design is outside the scope of the discussion for a MBus member node.

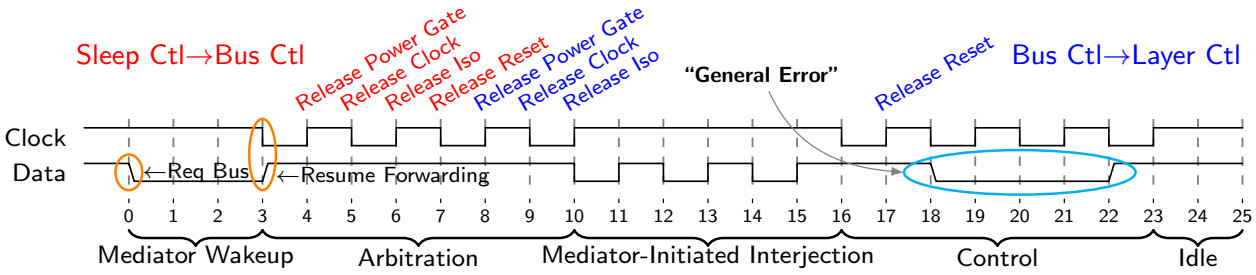


Figure 7: MBus Wakeup. Power-gated nodes repurpose the arbitration edges to wake the bus controller before data transmission starts. To self-wake, nodes can initiate a null transaction (shown here) by pulling down `DATA` and then resuming forwarding `DATA` before the arbitration edge. If no other node arbitrates, the mediator will detect no winner, raise a general error, and return the bus to idle. Arbitration produces enough edges to wake bus controllers before addressing. The null transaction produces enough edges to wake all of the MBus hierarchical power domains in a manner that is transparent to non-power-aware devices.

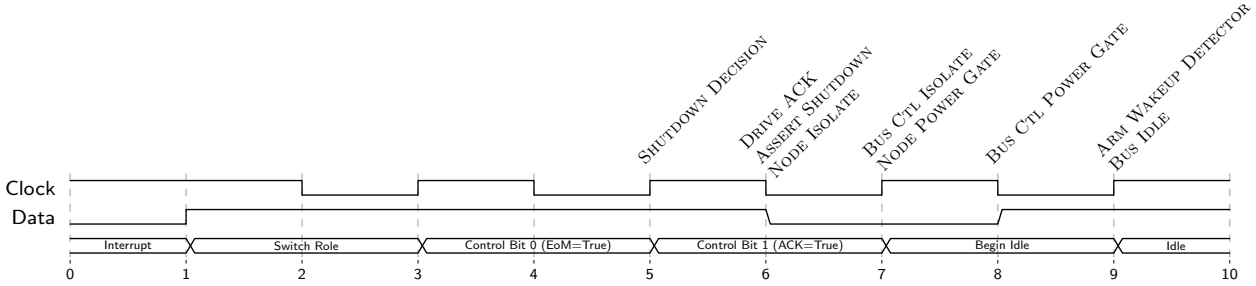


Figure 8: Shutdown Timing. The shutdown command is not confirmed until time 5 when the transmitter indicates a valid End of Message signal. At time 6, the bus controller acknowledges shutdown, asserts the `SHTDWN` signal to the sleep controller, and isolates the layer controller. At time 7, the sleep controller isolates the bus controller, and isolating the bus controller by definition power gates the layer controller. At time 8, the sleep controller power gates the bus controller, completing shutdown. At time 9 the bus is idle, and the sleep controller is waiting for the next wakeup.

4.6 The Mediator Node, In Brief

Explicitly not discussed in this document is any indication of how a power-gated mediator node is implemented. The period t_{long} is permitted to be arbitrary in length with the express intent of allowing sleeping mediator nodes to wake themselves using the falling edge of `DIN` as a trigger. The actual means for waking the mediator node are outside the scope of this document however.

Sleeping a mediator node is also outside the scope of MBus functionality. A mediator node must sample its `DIN` line upon returning to Idle (time 9 in Figure 8) by the rules laid out in 3.6 Return to Idle. If the `DIN` line is high at that time after an implementation-defined shutdown message, the mediator node may enter sleep.

5 Addressing Design

One of the MBus design points is to serve as a system interconnect for a physically constrained system. With that in mind, MBus attempts to optimize for a system of complex connected components. In particular, MBus expects that an individual member node may itself be composed of multiple functional units, which may each be individually addressed.

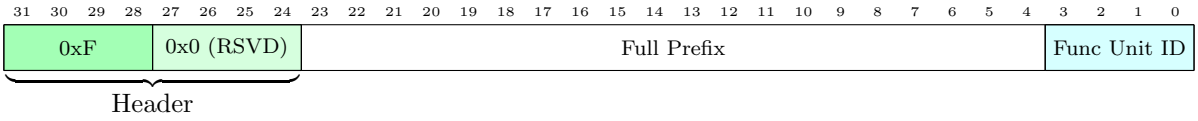
Supporting multiple, addressable functional units with only one set of exposed MBus signal pins could be done by instantiating several copies of a member node on each die, but this requires a bus interface module for every functional unit within an integrated chip. Instead, the bottom four bits of addresses are reserved to identify functional units. Nodes with more than sixteen functional units will require multiple addresses.

5.1 Address Types

MBus defines the term *prefix* to refer to the portion of the address that specifies which node is being addressed and reserves the term *address* to refer to a complete address—one that specifies a functional unit within a node.

Every MBus node has two prefixes, a *full* prefix and a *short* prefix. A short prefix is 4 bits long and a full prefix is 20 bits long. A short address is the composition of short prefix and a functional unit identifier. A full address is the composition of a header, a full prefix, and a functional unit identifier. To distinguish full and short addresses, the short prefix `0b1111` is reserved to identify a full address. The first four bits of the full address header is thus always `0b1111`.

5.2 Full Prefix Assignment

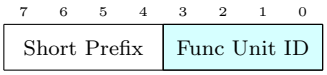


The purpose of full prefixes is to serve as a unique identifier for a node, akin to a product identifier. Full prefixes do not distinguish instantiations of a node, that is, multiple copies of a unique part will all have the same full prefix. This implies that multiple nodes in a single MBus instantiation may have the same full prefix. Bits 24-27 of the full address are reserved for other purposes (**?? ??**). The remaining range, bits 4-23, are available to be utilized as full prefixes. If a node has more than 16 functional units, it may have multiple full prefixes. These full prefixes should be sequential.

- The full prefix `0x00000` is reserved as the broadcast prefix.

The full prefixes ranging `0x00000-0x0000F` are reserved for legacy M3 devices. An allocation scheme for new full prefixes is currently undefined.

5.3 Short Prefix Assignment



The purpose of short prefixes is to uniquely identify nodes in an MBus system. Multiple nodes in a MBus instantiation **must not** have the same short prefix.

- The short prefix `0b0000` is reserved as the broadcast prefix.
- The short prefix `0b1111` is reserved to distinguish full addresses.

This leaves a remainder of 14 unique short prefixes. These short prefixes map to actual nodes instantiated in a MBus system. If there are multiple copies of the same node type (e.g. several external memory chips), each instance is given a unique short prefix. If a node has greater than 16 functional units it will be assigned multiple short prefixes, each mapping to one of the node's full prefixes.

In a MBus instantiation, short prefixes are assigned dynamically. Out of reset, all member nodes are assigned the short prefix Unassigned Short Prefix: `0b1111`. After system power-on, some node shall send

a series of Enumerate Node commands. MBus does not require that the mediator node send the Enumerate Node commands, however any node containing the mediator node should be capable of enumerating bus members. Any node capable of performing enumeration **must** be capable of snooping another node's performing of enumeration and **must** update its local enumeration state from the snooped data.

Short prefixes should be assigned in ascending numerical order beginning with prefix `0b0010`. It is the responsibility of the node(s) performing enumeration to ensure no duplicate short prefixes are assigned. As enumeration assignments are resolved by the topological priority arbitration, the short prefix assigned to a node should also correspond to its topological priority. This constraint may be violated in systems that do not enumerate prefixes in order, that use the Invalidate Prefix command to re-order short prefixes, or that use a Static Short Prefix Assignment where the statically assigned prefixes do not match the topology. Short prefixes cannot be used to affect priority. Priority is determined exclusively by physical topology.

Short prefix assignments **must** be preserved when nodes are powered gated.

5.3.1 Static Short Prefix Assignment

All MBus nodes **must** support dynamic short prefix assignments. A node may have a default short prefix assignment. Upon the receipt of any Enumerate Node commands, a node with a default short prefix **must** immediately invalidate its default short prefix and participate in the enumeration process. A node with a default short prefix that receives an Invalidate Prefix command shall set its short prefix to Unassigned Short Prefix: `0b1111`. A node with a default short prefix shall otherwise make no distinction from a dynamically assigned short prefix (e.g., it shall respond to a Query Devices command with its current short prefix, the static short prefix). An allocation scheme for default short prefixes is not yet defined.

6 MBus Protocol Design

To maximize device interoperability, MBus defines a higher-level protocol for basic point-to-point register and memory access. MBus also defines a protocol for broadcast messages.

MBus reserves two prefixes: a *broadcast* prefix and an *extension* prefix. The extension prefix is used in the short prefix space to identify full addresses. It is currently unused and reserved in the full prefix space.

6.1 Broadcast Messages (Address `0x0X`, `0xf000000X`)

MBus defines the broadcast short prefix as `0b0000` and the broadcast full prefix as `0x00000`. Broadcast messages are permitted to be of arbitrary length. Messages longer than 32 bits may be silently dropped by nodes with small buffers. A node **must not** interject a broadcast message to indicate buffer overflow. Interjections are permitted for broadcast messages greater than 4 bytes in length.

For broadcast messages, the functional unit ID field is used to define broadcast *channels*. Broadcast channel selection is used to differentiate between the different types of broadcast messages. MBus reserves half of these channels and leaves the rest as implementation-defined. The MSB of the broadcast channel identifier (address bit 3) shall identify MBus broadcast operations. If the MSB is `0` it indicates an official MBus broadcast message as specified in this document and subsequent revisions. Broadcast messages with a channel MSB of `1` are implementation-defined. It is recommended that nodes leveraging implementation-defined broadcast channels provide a mechanism to dynamically select broadcast channel to help mitigate conflicts.

A broadcast message that is not understood **must** be completely ignored. During acknowledgment, an ignorant node shall forward.

6.1.1 Broadcast Messages and Power-Gating

Some systems may have inter-node dependencies on `Layer` power state, e.g. activating a higher power regulator before a high-power component. For this reason, by default nodes must not change `Layer` power state upon receipt of a broadcast message, excepting messages that explicitly change `Layer` power state.

Some nodes, for example a general purpose processor that is snooping, may want to wake their `Layer` for all messages. This is permitted, but nodes with non-standard broadcast power behavior must clearly document power semantics to be MBus compliant.

6.1.2 Broadcast Channels and Messages

This section breaks down all of the defined MBus broadcast channels and messages. All undefined channels are reserved and shall not be used. A node receiving a broadcast message for a reserved channel shall ignore the message. It **must not** acknowledge a message on a reserved channel and **must** forward during the acknowledgment cycle.

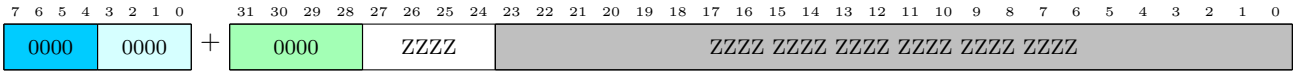
All MBus broadcast messages, except those sent on Broadcast Channels 2-7: Reserved, follow a common template. The messages are 32 bits long. The four most significant bits identify the message type/command. Some messages do not require all 32 bits. The unused bits are named *insignificant bits*. Messages may be truncated, omitting the insignificant bits on the wire[12].

All examples are shown with short addresses for space. There is no distinction between the use of the short or full broadcast address. Bitfields are presented **Address** + **Data**. Addresses are broken down into the Broadcast Prefix and the Broadcast Channel. Data is broken down into a Message Type Specifier and the message itself. In the bitfields, `0` and `1` indicate bits that must be set to that value, `X` indicates bits that depend on the current message, and `Z` indicates bits that should be *ignored*—accept any value, send as `0`. Insignificant Bits are also indicated as `Z`.

[12] With the caveat that all MBus messages must be byte-aligned. Some insignificant bits may still be sent on the wire as a consequence.

Broadcast Channel 0: Node Discovery and Enumeration Channel 0 is used for messages related to node discovery and enumeration. Channel 0 messages either require a response or are a response. Channel 0 response messages should not be sent unless solicited. The `Bus Controller` is responsible for handling Channel 0 messages. Channel 0 messages should not affect `Layer` power state.

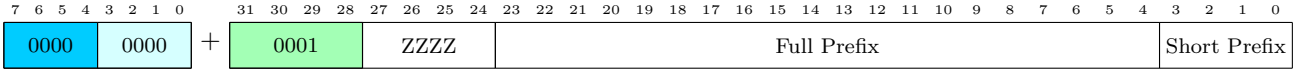
Query Devices



The query devices command is a request for all devices to broadcast their static full prefix and currently assigned short prefix on the bus. Every MBus node must prepare a Query/Enumerate Response when this message is received.

All nodes are required to support this message and respond.

Query/Enumerate Response



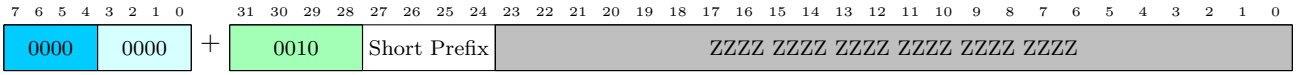
This message is sent in response to a Query Devices or Invalidate Prefix request. When responding to Query Devices, every node will be transmitting their address, and nodes should anticipate losing arbitration several times before they are able to send their response.

The top four bits of the data field identify the message as a Query Response. The next four bits are ignored. The following 20 bits contain the full prefix of the node. The final 4 bits are the currently assigned short prefix. Nodes that have not been enumerated should report a short prefix of `0b1111`.

This message must be sent in response to Query Devices or Enumerate Node. When responding to Query Devices, nodes **must** retry until the message is sent. When responding to Enumerate Node, nodes **must not** retry sending if arbitration is lost and **must** retry sending if interjected.[13]

All nodes are required to support this message.

Enumerate Node

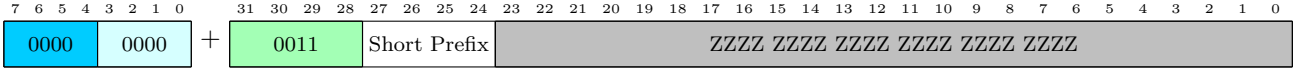


This message assigns a short prefix to a device. All nodes that receive this message and do not have an assigned short prefix **must** attempt to reply with a Query/Enumerate Response. Nodes with a short prefix shall ignore the message (broadcast NAK, that is, forward). Nodes shall perform exactly one attempt to reply to this message. The node that wins arbitration shall be assigned the short prefix from this message. Nodes that lose arbitration shall remain unchanged.

Nodes that have an assigned short prefix shall ignore this message.

All nodes are required to support this message and respond if appropriate.

Invalidate Prefix



This message clears the assignment of a short prefix. The bottom 4 bits specify the node whose prefix shall be reset. A node shall reset its prefix to Unassigned Short Prefix: `0b1111`. If the prefix to clear is set to Unassigned Short Prefix: `0b1111`, then all nodes shall reset their prefixes.

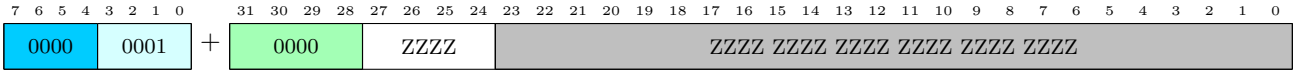
All nodes are required to support this message.

[13] An interjection should not occur during this message. Such an interjection would be an error.

Broadcast Channel 1: `Layer` Power Channel 1 is used to query and command the `Layer` power state of MBus nodes. Power-oblivious nodes may ignore channel 1. Power-aware nodes whose power model does not align well with these commands may ignore channel 1 messages *except* All Sleep. All nodes capable of entering a low-power state **must** enter their lowest power state in response to an All Sleep message.

A node `Layer` is implicitly waked when a message is addressed to it, explicitly issuing a wake command is unnecessary to communicate with a node. Nodes may build finer-grained power constructs beyond the macro `Layer` control provided by MBus. For the purposes of MBus, a node's "sleep" state should be a minimal power state. Nodes may have different sleep configurations, e.g. different interrupts that are armed.

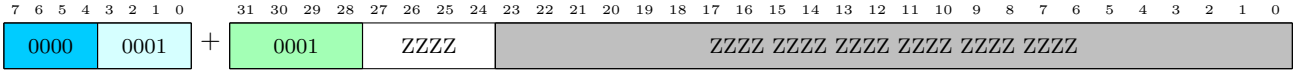
All Sleep



All nodes receiving this message **must** immediately enter their lowest possible power state. The bottom 28 bits of this message are reserved and should be *ignored*.

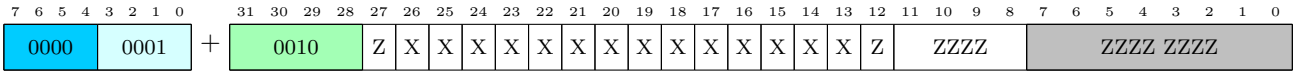
All power-aware nodes are required to support this message.

All Wake



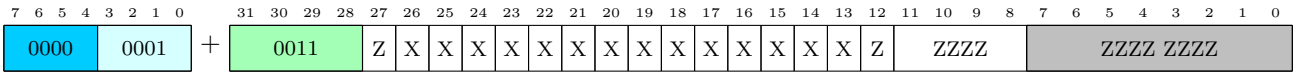
All nodes receiving this message **must** immediately wake up. The bottom 28 bits of this message are reserved and should be *ignored*.

Selective Sleep By Short Prefix



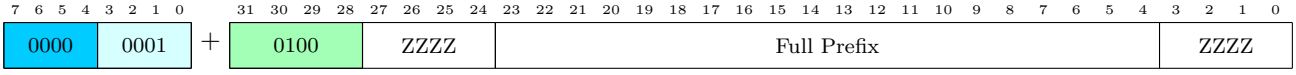
This message instructs selected nodes to sleep. The 16 bits of data are treated as a bit vector, mapping short prefixes to bit indicies. That is, the node with short prefix `0b1101` is controlled by the second bit received (bit 25 in the bit vector above). If a bit is set to `1`, the selected node **must** enter sleep mode. If a bit is set to `0`, the selected node should not change power state. The bits for prefixes `0b1111` and `0b0000` are ignored.

Selective Wake By Short Prefix



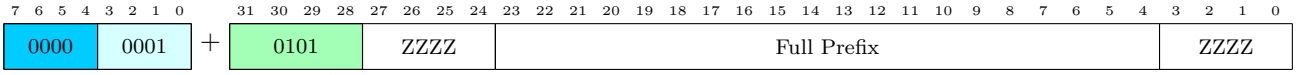
This message instructs selected nodes to wake. The 16 bits of data are treated as a bit vector, mapping short prefixes to bit indicies. That is, the node with short prefix `0b1101` is controlled by the second bit received (bit 18 in the bit vector above). If a bit is set to `1`, the selected node **must** wake up. If a bit is set to `0`, the selected node should not change power state. The bits for prefixes `0b1111` and `0b0000` are ignored.

Selective Sleep By Full Prefix



This message instructs selected nodes to sleep. Any node whose full prefix matches **must** enter sleep.

Selective Wake By Full Prefix



This message instructs selected nodes to wake. Any node whose full prefix matches **must** wake up.

Broadcast Channels 2-7: Reserved

6.2 Extension Messages (Address 0xffffffff)

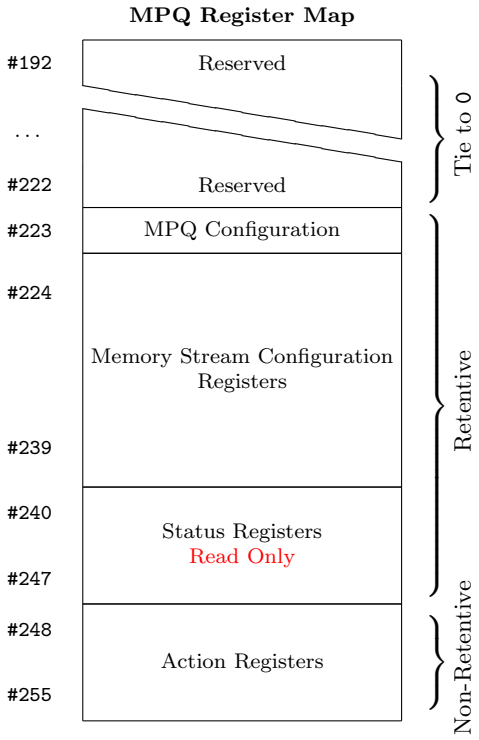
This address is reserved for future extensions to MBus.

7 MPQ: Point-to-Point Message Protocol

MBus also defines a common point-to-point messaging protocol: MPQ. Nodes are not required to support MPQ to be MBus compliant, however it is strongly encouraged.

MPQ defines two classes of data: register data and memory data. MPQ registers have 8 bit addresses and are 24 bits wide. MPQ memory has 32 bit addresses and stores data that is 32 bits wide. The register address space and memory address space are separate constructs, that is register address `0x00` need not map to memory address `0x00000000`, although aliasing is permitted.

7.1 MPQ Registers (Reg #192–255)



MPQ reserves the top 64 registers for control and configuration.

This space configures various MPQ capabilities. Much of the space is currently reserved for future expansion.

Attempts to write configuration for unsupported features (e.g. configuration of a memory streaming channel on a device with no memory) is undefined. Attempts to read unsupported features **must** NAK or return all `0`.

This section documents each of the registers. The top two bits of each MPQ register address are `11` followed by six bits that identify the command type. All reserved bits should be treated as RZWI.

7.1.1 Reserved Registers

All registers not specified here are reserved. Writes to reserved registers should be ignored. Reads from reserved registers should return all `0`.

7.1.2 Reg #223: MPQ Record and Interrupt Control

Default: `0x0000XX`

Each time a node completes a MPQ command, it checks the associated command `INT` bit. If high, the layer owner is interrupted.

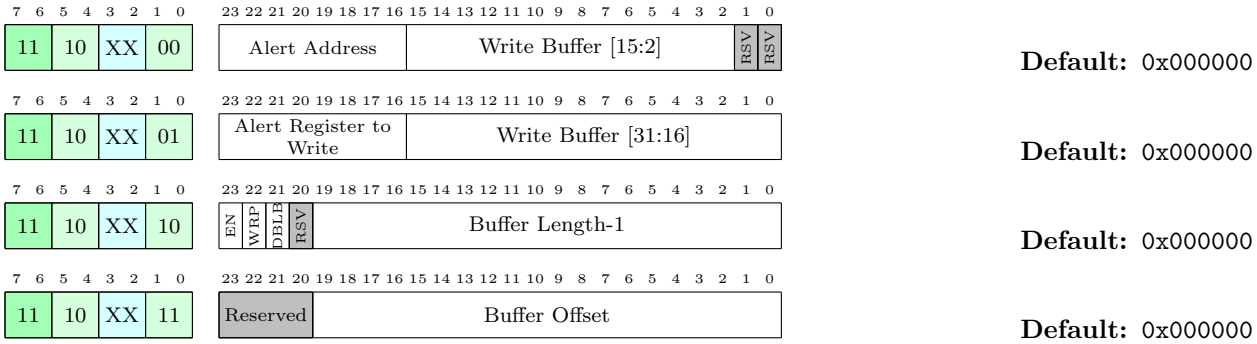
Default Value: The default value is implementation dependent. Default on for CPU, off for others?

TODO: Do we care about **Record** if `INT` is low? Separate control?

TODO: For read requests does it make more sense for the interrupt to stop the request and pass it to the layer owner instead? How does this interact with the various `EN` bits?

TODO: What are the ways in which a command that comes in the interrupt port from the layer should differ from a command that comes from the bus? Separate interrupt control?

7.1.3 Reg #224–239: Memory Stream Configuration



In MPQ each node has up to four independent, identical memory streaming channels. Each channel has two configuration registers. The two registers work together to configure each channel.

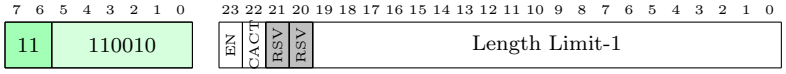
- {R01[15:0], R00[15:2], 2'b00}: Write Buffer.
  - Pointer to the beginning of a buffer in memory.
- {R00[23:16]}: Alert Address.
  - Defines alerts are sent. If the alert prefix (bits 23:20) are set to the full-prefix indicator 1111, the alert is suppressed.
  - Alerts are sent whenever the end of the buffer is reached. If DBLB is active, an alert is also sent when the halfway point of the buffer is reached.
  - When a memory stream alert occurs, a node sends the following message:

    - The Alert Address is specified by bits 23:16 in R00.
    - The top 8 bits of data are are specified by bits 23:16 in R01.
    - The Alerting Stream Channel is made up of this node's short prefix, followed by 01, followed by the channel that generated the alert—this should be the same address used to write to this stream channel.
    - The EN bit reports the current state of the EN bit of the alerting channel when the alert was sent.
    - The WRP bit is set if the write that generated this alert reached the end of the stream buffer.
    - The DBLB bit is set if the write that generated this alert reached the halfway point of the stream buffer and double-buffering is active for this stream.
    - The OVFL bit is set if the write that generated this alert reached the end of the stream buffer and there was already a pending alert with the WRP bit set or if the write that generated this alert reached the halfway point of the stream buffer and double buffering is active for this stream and there was already a pending alert with the DBLB bit set.
  - TODO: What happens if a node wishes to alert itself? e.g. a CPU that puts itself to sleep but wants to be interrupted when 1,000 samples have been written to memory?
- {R10[19:0]}: Buffer Length-1.
  - Defines the length of the buffer.
- {R10[23]}: Enable (EN).
  - Controls whether this channel is enabled. If EN is 0, a node must not modify memory in response to a memory stream message.[14]
  - **Acknowledement/Interjection:** The behavior of a stream write when EN is 0 is undefined. Receivers are encouraged to interject and indicate receiver error, however they may exhibit any behavior, including ACK'ing the transaction and silently ignoring it.
- {R10[22]}: Wrap (WRP).

[14] Implementation Tip: Any undefined MPQ registers are defined to be RZWI, that is a read from an undefined register will read as all 0's (and a write ignored). Upon a read, this will return 0 for EN as required. Nodes that do not implement a memory stream channel do not require any logic to handle any memory stream messages, simply leave the register undefined.

  - Defines node behavior when the end of the buffer is reached. If `WRP` is high, the `Write Address Counter` should reset to its original value. If `WRP` is low, the `Write Address Counter` value should be unchanged (it should thus be one past the end of the valid buffer) and `EN` should be set to `0`.
- {`R10[21]`}: `Double Buffer (DBLB)`.
  - Controls double-buffering mode. If double-buffering is active, the node should generate an alert halfway through the buffer in addition to at the end of the buffer. This mode is most useful when combined with `WRP`.

7.1.4 Reg #242: Bulk Memory Message Control

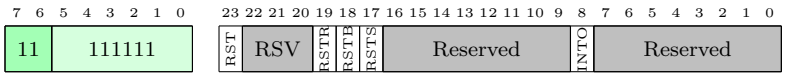


Def: `0x800000`, `0x000000` if no mem

The register controls the response of this chip upon the receipt of a Memory Bulk Write command.

- {`R[23]`}: `Enable (EN)`.
  - Controls whether bulk memory transactions written to this device are enabled. If `EN` is `0`, a node must not modify the contents of memory in response to a bulk memory transaction.
  - **Acknowledement/Interjection:** The behavior of a bulk write when `EN` is `0` is undefined. Receivers are encouraged to interject and indicate receiver error, however they may exhibit any behavior, including ACK'ing the transaction and silently ignoring it.
- {`R[22]`}: `Control Active (CACT)`.
  - If this bit is high, this register's length field acts as a limit for the maximum permitted bulk message length. A bulk message is allowed to write until this message limit is reached. If more data comes, the message **must** be NAK'd. The receiver should interject with receiver error as soon as it knows the length limit has been exceeded.
- {`R[19:0]`}: `Length Limit`.
  - The maximum permitted length in words of a memory bulk write message to any address.

7.1.5 Reg #255: Action Register



<No Storage>

This register requests that an action be performed. It is an error to request more than one action in a single request. Actions are processed from MSB to LSB, that is, if more than one action is requested *only* the highest priority action is actually taken.

A read from this register shall always return all `0`.

A write of all `0` to this register shall perform no actions.

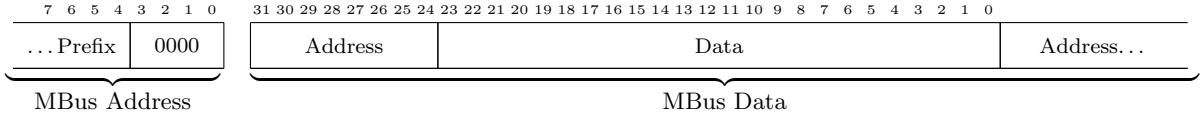
If any bit written to this register is non-zero, writing this register **must** be the last operation in the transaction. The behavoir of anything after a register action request in the same transaction is undefined.

- {`R[23]`}: `Reset (RST)`.
  - Reset the entire node. The exact result of a reset is implementation-defined, however this request should be the most aggresive form of reset available.
- {`R[19]`}: `Reset Register Registers (RSTR)`. FIXME
  - Reset MPQ configuration registers that affect register protocol behavior to their default value.
  - Resets #240.
- {`R[18]`}: `Reset Bulk Registers (RSTB)`.
  - Reset MPQ configuration registers that affect memory bulk transfers to their default value.
  - Resets #242.
- {`R[17]`}: `Reset Stream Registers (RSTS)`.
  - Reset MPQ configuration registers that affect memory stream transfers to their default value.
  - Resets #224–239.
- {`R[8]`}: `Interrupt Owner (INTO)`.
  - Asserts the Layer Owner Interrupt.

7.2 Register Commands

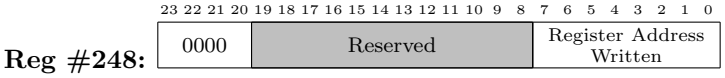
MPQ register space is an 8 bit addressable array of 24 bit wide registers. Any undefined bits are treated as RZWI (Read as Zero, Write Ignored).

7.2.1 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.

Record: Upon command completion, write local registers:

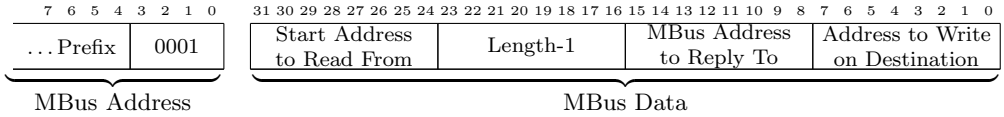


Overflow: If more data is received after writing the last register (`0xff`), the destination address wraps and registers continue to be written, beginning at address `0x00`. *Tests:* **??**.

Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Interjection 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 interjection, only the first command was applied. If 64 bits are received, the first two commands are applied. *Tests: A.3.1, A.3.2.*

7.2.2 Register Read

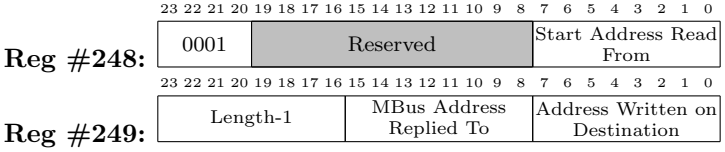


Bits 24-31 specify the address of the register to be read. Bits 16-23 may be used to request that multiple registers are sent. This field is a count of the number of values to be sent less one, that is a value of 0 requests 1 register is read and a value of 255 requests that all 256 registers are sent. Bits 8-15 are the MBus address the reply is sent to. Bits 0-7 specify the first address field of the Register Write response.

The response message is sent to the MBus 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. For reads of more than one register, the address field in the response is incremented by 1 for each register.

The response will always respond with the requested number of registers. If a request for register #256 would have been made, the request wraps and begins from register #0. The destination register address wraps similarly.

Record: Upon command completion, write local registers:



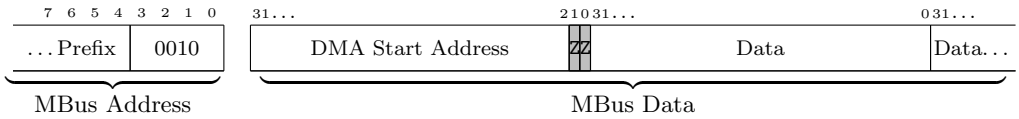
Note: The MBus address to reply to **must** be copied exactly. The FU_ID is not required to be Register Write. For example, if the FU_ID is Memory Stream Write, the effect is dumping the current register state to memory on the target addresss. *Tests: A.1.1.*

Interjection Semantics: If the reply is interjected, the transaction is aborted and is **not** retried.

7.3 Memory Commands

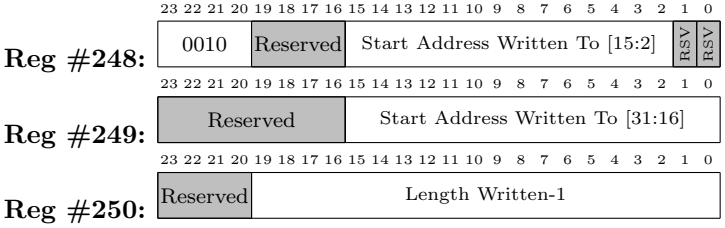
MPQ memory space is a 32 bit addressable array of 32 bit words of memory. Any undefined accesses are treated as RZWI (Read as Zero, Write Ignored). MPQ provides two types of memory commands: bulk and stream. A bulk memory transaction is a wholly self-contained event designed for DMA of large blocks of memory. Memory streams pre-configure the stream destination address and omit it from subsequent transactions, relying on the receiver to maintain and increment a destination pointer. Streams are useful for applications such as continuous sampling, where multiple, short messages are generated.

7.3.1 Memory Bulk Write



The first word received is the address in memory to begin writing to. The bottom two bits of the address field are reserved 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.

Record: Upon command completion, write local registers:



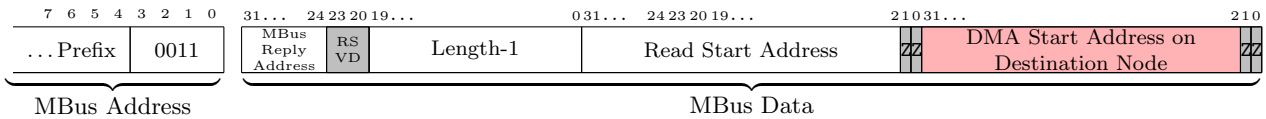
Implementation Note: These registers may be updated while the command is running, so long as they have the correct value once it completes. They are a good place to store the pointer and counter while the command is active instead of instantiating dedicated counter and address registers.

Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Overflow: If more data is received after writing the last address in memory (`0xfffffffc`), the destination address wraps and data continues to be written, beginning at address `0x00000000`. *Tests: A.2.1.*

Interjection 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 interjection, only the first word was written to memory. If 96 bits are received, the first two words were written to memory. *Tests: A.3.3, A.3.4.*

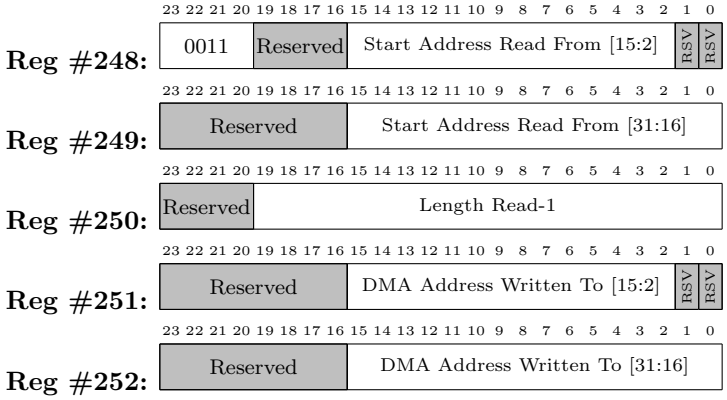
7.3.2 Memory Read



The first word indicates the MBus 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. The second word is the address in memory to read from. The bottom two bits of the address field are reserved and **must** be transmitted as 0.

The third word is **optional**. If the third word is present, it is prepended to the reply (generating a message with a Memory Bulk Write formatted payload). If the third word is omitted, data is immediately placed on the bus (generating a message with a Memory Stream Write formatted payload).

Record: Upon command completion, write local registers:

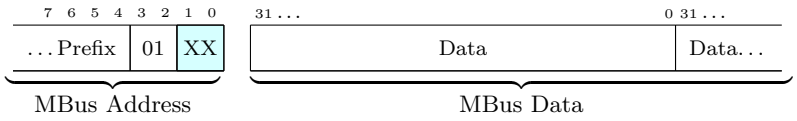


TODO: Message length isn't passed here, how to know if target address is included?

Overflow: 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 wraps and continues sending from address `0x00000000`. *Tests: A.2.1.*

Interjection Semantics: If the reply is interjected, the transaction is aborted and is **not** retried.

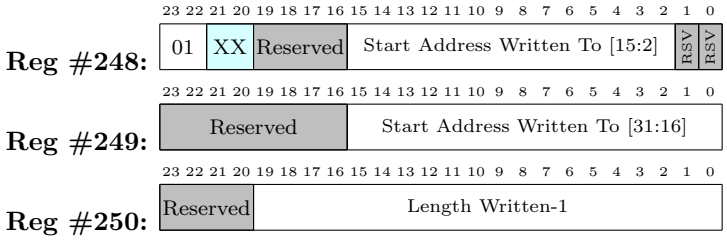
7.3.3 Memory Stream Write



MPQ nodes have up to four streaming memory channels. Each channel is controlled by configuration registers (Reg #224–239: Memory Stream Configuration). The destination of a memory stream write is specified by a combination of channel selection—the last two bits of the FU_ID—and the pre-arranged configuration.

The message payload is all data. The destination address is automatically incremented every time a word is written. There is no limit on the length of this message.

Record: Upon command completion, write local registers:



Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Overflow: If more data is received after writing the last address in memory (`0xfffffffc`), the destination address wraps and data continues to be written, beginning at address `0x00000000`. *Tests:* **??**.

Interjection 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 interjection, only the first word was written to memory. If 96 bits are received, the first two words were written to memory. *Tests:* **??**, **??**.

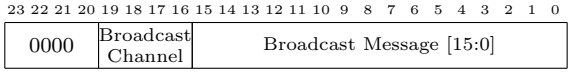
7.4 Broadcast Snooping

Broadcast Channel 0: Node Discovery and Enumeration and Broadcast Channel 1: `Layer` Power are built into the MBus protocol which runs below MPQ. However, it is possible that a MPQ node may wish to snoop broadcast traffic (in particular, Query/Enumerate Response).

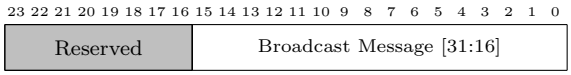
If broadcast snooping is active TODO: Configuration register for this. Possibly some other filters such as channels?, whenever a broadcast message is received:

Record: Upon broadcast reception, write local registers:

Reg #248:



Reg #249:



TODO: Splitting words across two 24-bit registers all of the time is getting a little annoying. I'm not really interested in re-visitng the 24-bit register decision; there are a number of reasons that was a good choice. I am curious, however, if we can do something intelligent that maps MMIO accesses from the CPU intelligently so that every MMIO read isn't wasting 8 bits reading nothing.

7.5 Undefined Commands

If command with an unknown FU_ID is received, MPQ behavior is completely undefined.

8 MPQ Programmer's Model

In addition to MPQ, MBus also defines a suggested programmer API. While MPQ implementations are not required to adhere to this specification, it is highly recommended. This specification includes a lightweight software library built on top of this API.

asynchronous	Dropped message notification.
asynchronous	Notification of received message and type.

8.1 Overview, Memory Map, and Interrupts

MPQ requires a 2^{24} region of memory mapped I/O space. By default, MPQ uses `0xA5000000-0xA6000000`, however this may be changed by defining the macro `MPQ_MMIO_PREFIX` before including `mpq.h`. 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`	*synchronous*	Interrupt Configuration [Read/Write]
`0xA51`	`R`	*synchronous*	??
`0xA52`	`W`	*asynchronous*	Arbitrarily Long Write [Write Only]
`0xA53`	`R`	*synchronous*	Transaction Result [Read Only]
`0xA55`	`RW`	*synchronous*	RX Buffer (multi-word) [Read/Write]
⋮			*Reserved*

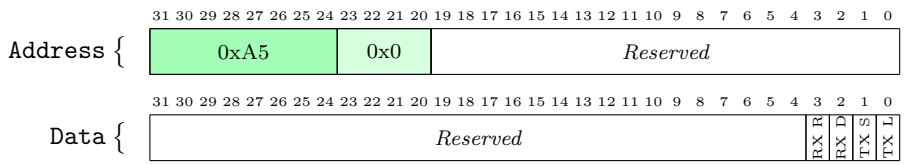
In the following sections, there are many reserved bits. These bits shall be considered RZWI (Read as Zero, Write Ignored). These bits are currently unused but are reserved for possible future use. Software should write `0` to reserved bits to ensure future compatibility.

8.2 Configurable Parameters

8.2.1 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 8.4.1 {RX D}
1	Bus received a message 8.4.1 {RX R}



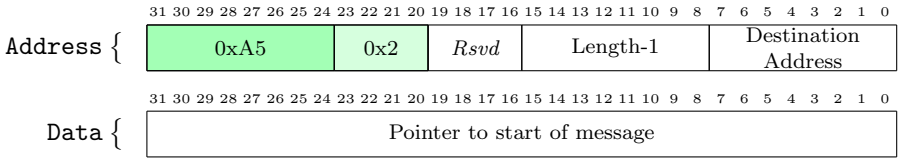
8.3 TX Transaction Registers

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.

8.3.1 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



8.3.2 Transaction Result [Read Only]

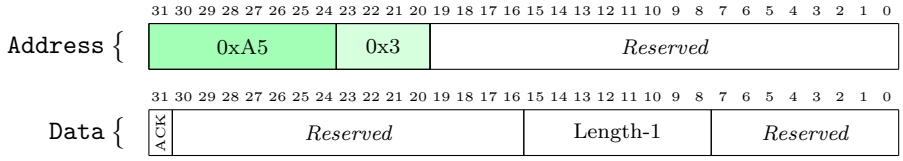
A register that stores the result of the previous transaction request. "Result" is defined as the number of bytes successfully sent on the bus before a reset event occurred and whether the transaction was ACK or NAK'd. Note that the ACK bit will only be set if the entire transaction completed successfully 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 **??** 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

8.4 Transaction RX

8.4.1 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 8.2.1.

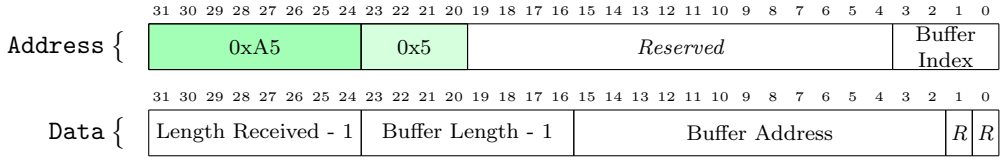


RX Received A message has been received. Depending on the expense of interrupts (TODO **??**), this may be a unique interrupt per RX Buffer or a single shared interrupt with a register to read where a message came in. This interrupt may be suppressed by 8.2.1. **Disabling this interrupt will disable message reception.** The bus controller will NAK even single-word messages if this interrupt is disabled.

8.4.2 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



9 Specifications

9.1 Granularity

MBus sends data in units of 8 bit bytes. Transmissions must be modulo 8 bits in length, that is they must end on byte boundaries (for rationale, see Granularity).

9.2 Endianness: Byte-Level

MBus sends from Byte 0...Byte N.

9.3 Endianness: Bit-Level

MBus sends bytes most significant bit first (bit 7...0).

9.4 Minimum Message Length

The minimum legal message on MBus is *zero* bits long. A zero length message can be used to query whether a device is present on the bus. A node receiving a zero length message addressed to it **must** ACK the message. Whether higher layers are indicated of a query message is implementation dependent.

See 9.12 Interjection Rules for details on the minimum uninterruptable message length.

9.5 Minimum Maximum Message Length

All MBus mediators must permit a minimum of 1 K (1024 bits) of data to be transmitted before aborting a transaction due to a hung transmitter.

The message length counter shall count the number of positive clock edges received after Begin Transmission at the mediator node's `CLK_IN` port up to and including the last bit latched (Request Interjection in **??**). The counter shall be an *inclusive* counter, that is the mediator node shall not interject until it receives the $N + 1$th bit.

Commands to query and configure the upper bound are specified in Broadcast Channels 2-7: Reserved.

9.6 Retransmission

There is no hardware re-transmission primitive. The hardware provides a transaction-level message acknowledgment, indicating whether the complete message was received or not. Retransmission of failed messages is left to software.

9.7 Minimum Buffer Size

All MBus nodes are required to support both short and full addresses (9.10 Addressing). All MBus nodes are further required to accept a minimum of 32 bits (4 bytes) of data in an individual transmission.

9.8 Buffer Overflow / Flow Control

On the average, the expectation in a MBus system is that a transmitting node knows the capacity of the receiving node. If a receiving node does not have enough space for the current transmission, it **must** interject the current transmission and indicate a receiver error (Control Bits: `01`).

The receiver must interject as soon as it is *certain* that it has exceeded its receive capacity. This detection must be performed carefully and must take into account that two bits beyond the complete transmission may be temporarily received (as is the case for Node 1 in **??**). The receiving state machine should not transition into DECIDED_TO_INTERJECT until the 3rd bit of a byte that it does not have space for. The receiver may wait until the 8th bit to transition its state machine to request interjection, but no later than that such that it is assured that it will interject the transmission before the transmitter attempts to interject to signal end of message even if the receiver is of lower priority. XXX This should be a specified test

9.9 Clock Speed

TODO: What is appropriate specification for clock speeds? I2C has defined 'speed-classes'; SPI is a free-for-all; UART & co has bauds...

9.10 Addressing

MBus defines two types of addresses: *short* 8-bit addresses and *full* 32 bit addresses. No short address may begin with `1111` and all full addresses must begin with `1111`. Address assignment is laid out in 5.3 Short Prefix Assignment and 5.2 Full Prefix Assignment respectively. Addresses `0x00`, `0x01`, and `0xffffffff` are reserved.

9.11 Unassigned Short Prefix: `0b1111`

Out of reset, nodes self-assign a short prefix of `0b1111`. This short prefix shall be considered as a sentinel value where short prefixes are reported, indicating that the node does not currently have a short prefix assigned.

9.12 Interjection Rules

MBus permits any node to interject any message for any reason. To allow for minimal forward progress, however, MBus requires that nodes permit a minimum of 32 bits (4 bytes) be transmitted without interjection. An exception is made for the transmitting node, which is permitted to send messages shorter than 32 bits.

Nodes wishing to interject must wait until either 41 or 65 clocks after Begin Transmission to interrupt the bus (depending on whether the current message was addressed to a short address or full address). The interjecting node must wait until it has latched the 33rd bit of data before attempting to interject.[15]

9.13 Failed Arbitration (Spurious Wakeup)

During normal arbitration, when the arbitration is resolved the mediator node's `DIN` line should be low no matter who is participating. If the mediator node finds its `DIN` line is high on the arbitration edge, it would indicate that no one is participating. The mediator node must treat this as an error and reset the bus.

The mediator node **must not** immediately interject the bus, however. As laid out in 1.2 Power Gated Nodes, the arbitration edges may be used as input to node sleep controllers. To avoid potential issues related to half-waking power gated nodes, the mediator node must wait until Begin Transmission before interjecting the bus.

The control bits **must** be driven by the mediator node. They **must** be `00`, general error.

[15] If the node instead attempted to interject on the 32nd bit and was a topologically higher-priority node, it would not permit the transmitter to signal End of Message.

10 ToDo

10.1 Future Extensions

There are properties that would be nice to have, but introduce greater complexity. While they are not in the current design, some methods for designing them are considered here such that they are still feasible for future implementation in a backwards-compatible manner.

10.1.1 Automatically fragment long MPQ messages

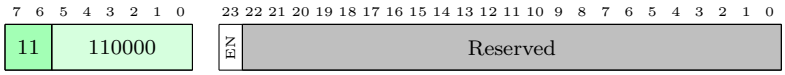
Add a configuration register for a maximum message length, maybe one for each message type, that limits maximum transaction length. Requests to send messages over this length should be automatically fragmented by the sender.

10.1.2 Full Address Support in MPQ

Currently there is no way to generate a response that sends a message to a full address. A full prefix is 24 bits long. The following are some possible methods to support full addresses:

1. **Use new FU_IDs.** Easiest solution, but uses a lot of the remaining FU_IDs. Could do something creative such as the full prefix (24 bits) is sent and then the next 3 or 4 bits map the remaining words onto one of the existing FU_IDs. This wastes at least the 4 bits that previously specified the short prefix.
2. **Full Prefix Register(s).** Store a full prefix in a register. Since registers are also 24 bits, they can hold a full prefix exactly. This is actually a little unfortunate, since a destination FU_ID also needs to be specified. While the short prefix `1111` in the current message can be used as a sentinel, something needs to indicate which register the full prefix should be read from. The FU_ID field could be repurposed to this end, but then there is nothing to specify the FU_ID of the eventual message. This could spill over to two registers, but that is an inefficient use of registers. Full prefix registers also become another contested, shared resource with this scheme.
3. **Optional Extra Word.** Use the existing short prefix as a sentinel. If set to `1111`, the (next, last) word will be the full prefix. Since the short prefix is the first 4 bits, could also just "replace" the first word of the message with a full address—this has nice symmetry with how short/full prefixes are interpreted on the bus. In any of these cases, an extra 8 bits become available for some other purpose. The runtime variable length message can be tricky to implement. Injecting an extra word into the middle of a message makes it harder for simple devices with fixed send buffers to support these messages.

10.1.3 Reg #240: Register Message Control



Default: `0x800000`

- {R[23]}: Enable (EN).

  Controls whether register writes to this device are enabled. If `EN` is `0`, attempts to write registers #0–247 are silently ignored.

  Note: Registers #248–255 (`b'11111XXX`) can always be written.

  Acknowledement/Interjection: Since a single register write transaction could write both enabled and disabled registers, a node should not interject when a disabled register is written. A node should ACK to indicate that the transmission was received successfully, even if nothing is actually written. *This behavior is subject to change in future revisions of MPQ.*

  Warning: If `EN` is disabled, it cannot be re-enabled by a write to this register since register writes are disabled. Access can be recovered using Reg #255: Action Register, however this will reset all registers.

  TODO What about a local write (e.g. a register write via an interrupt from the CPU)? Should probably always allow those. Does that make sense as an exception?

11 Document Revision History

- Revision 0.4 (rXXXX) –
  - Pull in protocol documentation to this specification
  - Register read always returns length requested, wrapping if needed
  - Memory bulk transfer always returns length requirements, wrapping if needed
  - Add start address to register read
  - Add DMA start address to memory bulk read
  - Add memory stream protocol
  - Add tests appendix
  - Simplify interaction of broadcast messages and power states
  - Remove specification of broadcast channels 2 and 3
  - Interrupt → interjection
  - Master → mediator
  - Pull MPQ programmer model into this spec
- Revision 0.3 (r7650) – April 18, 2013
  - Add full M3 team to the authors list
  - Add reserved edges to arbitration
- Revision 0.2 (r4516) – May 6, 2013
  - Add Broadcast Channel 3
  - Clean up some of the more dated requirements text
  - Terminology change, “control” to “master” node
- Revision 0.1 (r4333) – Apr 22, 2012
  - Initial revision

A Test Cases

This section documents some of the trickier corner cases in MBus design. This is by no means a comprehensive list of tests, but each of these cases should be tested to ensure compatibility across MBus implementations.

For most tests, we recommend that nodes are be assigned a Static Short Prefix Assignment so that enumeration can be skipped.

A.1 Registers

A.1.1 Dump Register Content to Memory and Restore to Other Registers

```
A) 4 Nodes:  N1 -> N2 -> N3 -> MED (-> N1)
```

This test dumps a node's registers to the memory of another node by requesting a register read that issues a streaming memory write. Some of these dumped registers are then written to different registers on another node by using a streaming memory read that issues a register write.

1. Begin with N1's register data full of known, non-zero values. N2 and N3 registers and memory should be zeroed.
2. N2 configures the memory stream address for N3:
   ??: 0x3X + 0xXXXXXXXX
3. N2 requests that N1 copy 19 registers to N3's memory. Choose register addresses such that both read and destination addresses wrap.
   ??: 0x11 + 0xED_12_3X_0xF8
4. Verify the memory contents of N3.
5. N1 requests that 9 of the registers are written to N2:
   ??: 0x??

A.2 Memory

A.2.1 Bulk Transaction Overflow Wrapping

```
A) 3 Nodes:  N1 -> N2 -> MED (-> N1)
```

This test verifies both basic memory bulk transfer operation and that the bulk transfers handle overflow wrapping correctly.

1. Begin with N1's memory full of known, non-zero values. N2 memory should be zeroed.
2. N1 sends a request to N2 to read the last word and first 3 words of its memory and writes it to the last 2 words and first 2 words of N1's memory.
   ??: 0x25 + 0xfffffffc_12_000011_ffffffff8
3. Verify the correct values from N2 are written to memory in N1.

```
Test Name:            Success:
mem-bulk-overflow-A - Write 4 words to N2
```

A.3 Interjection Timing

A.3.1 Third-Party Interjector for Many Registers

```
A) 4 Nodes:  RX -> INJ -> TX -> MED (-> RX)
```

Have the TX node send a register write command that writes at least 3 registers long. The INJ node should generate an interjection on the 63rd bit of **data** with control bit 0 set to `!EoM`. The effect should be that the transmitter sees 63 data clock edges and the receiver sees 65 data clock edges. At the end of the transaction, the first register should be written, the second register should not. The TX layer controller should believe that the transmission failed and it has written either 7 bytes or 1 word (depending on implementation), but not 8 bytes or 2 words.

Repeat this test, interjecting on the 64th edge. The result should be the same.

```
B) 4 Nodes:  TX -> INJ -> RX -> MED (-> TX)
C) 4 Nodes:  TX -> RX -> INJ -> MED (-> TX)
```

Repeat the same test with these node positions.
Repeat these tests, interjecting on the 64th edge. The result should be the same.

```
D) 4 Nodes:  INJ -> RX -> TX -> MED (-> INJ)
```

Repeat the same test but interject at the 64th, 65th, and 66th data edges. For 64 and 65, the result should be the same. For 66, two registers should be written.

```
Test Name:          Success:
inj-reg-long-A-63 - Write 1 register
inj-reg-long-A-64 - Write 1 register
inj-reg-long-B-63 - Write 1 register
inj-reg-long-B-64 - Write 1 register
inj-reg-long-C-63 - Write 1 register
inj-reg-long-C-64 - Write 1 register
inj-reg-long-D-63 - Write 1 register
inj-reg-long-D-64 - Write 1 register
inj-reg-long-D-65 - Write 1 register
inj-reg-long-D-66 - Write 2 registers
```

A.3.2 Third-Party Interjector at Last Register

This test is the same as the previous, except the TX node should attempt to write exactly 2 registers.

```
Test Name:         Success:
inj-reg-end-A-63 - Write 1 register
inj-reg-end-A-64 - Write 1 register
inj-reg-end-B-63 - Write 1 register
inj-reg-end-B-64 - Write 2 registers
inj-reg-end-C-63 - Write 1 register
inj-reg-end-C-64 - Write 2 registers
inj-reg-end-D-63 - Write 1 register
inj-reg-end-D-64 - Write 1 register
inj-reg-end-D-65 - Write 2 registers
inj-reg-end-D-66 - Write 2 registers
```

A.3.3 Third-Party Interjector for Long Memory Bulk Transfer

Repeat the same tests as Third-Party Interjector for Many Registers, adding an address to the MBus data payload and interjecting 32 cycles later such that either one or two words are written to memory.

```
Test Name:          Success:
inj-mem-bulk-long-A-63 - Write 1 word
inj-mem-bulk-long-A-64 - Write 1 word
inj-mem-bulk-long-B-63 - Write 1 word
inj-mem-bulk-long-B-64 - Write 1 word
inj-mem-bulk-long-C-63 - Write 1 word
inj-mem-bulk-long-C-64 - Write 1 word
inj-mem-bulk-long-D-63 - Write 1 word
inj-mem-bulk-long-D-64 - Write 1 word
inj-mem-bulk-long-D-65 - Write 1 word
inj-mem-bulk-long-D-66 - Write 2 words
```

A.3.4 Third-Party Interjector at End of Memory Bulk Transfer

Repeat the same tests as Third-Party Interjector at Last Register, adding an address to the MBus data payload and interjecting 32 cycles later such that either one or two words are written to memory.

```
Test Name:         Success:
inj-mem-bulk-end-A-63 - Write 1 word
inj-mem-bulk-end-A-64 - Write 1 word
inj-mem-bulk-end-B-63 - Write 1 word
inj-mem-bulk-end-B-64 - Write 2 words
inj-mem-bulk-end-C-63 - Write 1 word
inj-mem-bulk-end-C-64 - Write 2 words
inj-mem-bulk-end-D-63 - Write 1 word
inj-mem-bulk-end-D-64 - Write 1 word
inj-mem-bulk-end-D-65 - Write 2 words
inj-mem-bulk-end-D-66 - Write 2 words
```

B Scratchpad

This section contains ideas for future additions to the MBus protocol. Items in this section are **not** stable and should **not** be implemented in current MBus designs.

Broadcast Channel 2: MBus Configuration The purpose of this channel is to configure any MBus parameters. Commands issued on channel 2 **must** be targeted for the mediator node. By utilizing a broadcast channel all interested nodes can easily "subscribe" to configuration messages. Using a broadcast channel also permits nodes to hard-code the address for configuration messages.

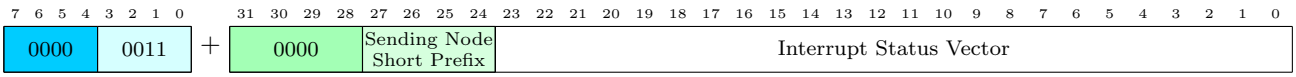
The current MBus specification does not define any channel 2 messages. Possible future messages include:

- Maximum message length
- Clock speed

Broadcast Channel 3: Member Node Events The purpose of this channel is for broadcast dissemination of events that are specific to one member node. It also permits the simplification of a member node design by permitting the member node to hard-code a destination address for certain classes of messages (e.g. interrupts).

Member node event messages **must** also include a Member Node Identifier—the sending node's current short prefix—to disambiguate member node events. It is possible, and acceptable, for a member node to generate a member node event message before enumeration has completed, in which case nodes without a Static Short Prefix Assignment send their current short prefix Unassigned Short Prefix: `0b1111`.

Member Node Level Interrupt

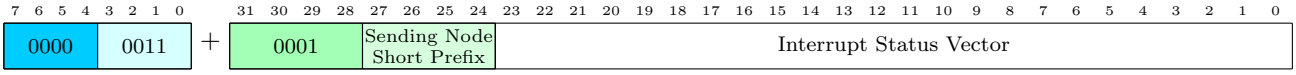


This message announces that an interrupt has occurred on this member node. The semantics of a MBus level interrupt dictate that an interrupt must be explicitly cleared. A member node **must not** generate another level interrupt message *for the same interrupt* until that interrupt is explicitly cleared. A *different* interrupt may occur before the first interrupt is cleared, in which case the bit vector **must** indicate both interrupts as active. Member nodes are permitted to batch or delay interrupts if appropriate, that is, a single level interrupt message may indicate the multiple interrupts have occurred.

Clearing interrupts is **not** a Channel 3 message, as doing so would require all member nodes to wake for every Channel 3 message. As a consequence, systems with multiple nodes capable of clearing a member node interrupt should develop another means of coordinating the responsibility. A member node designed to have multiple potential controlling nodes should consider defining a broadcast control channel.

MBus defines active to be logical `1`. Member nodes whose internal interrupts are active low must invert the signal before broadcasting the interrupt vector. The remaining 24 bits of this message are defined as a bit vector representing the current status of 24 interrupts.

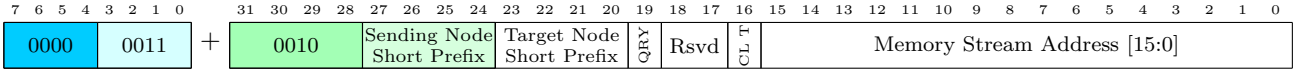
Member Node Edge Interrupt



This message announces that an interrupt has occurred on this member node. The semantics of a MBus edge interrupt dictate that the interrupt is implicitly cleared. That is, after the broadcast message announcing the interrupt has been sent the same interrupt is eligible to fire again. Member nodes are permitted to batch or delay interrupts if appropriate, that is, a single edge interrupt message may indicate multiple interrupts have occurred.

MBus defines active to be logical `1`. Member nodes whose internal interrupts are active low must invert the signal before broadcasting the interrupt vector. The remaining 24 bits of this message are defined as a bit vector representing the current status of 24 interrupts.

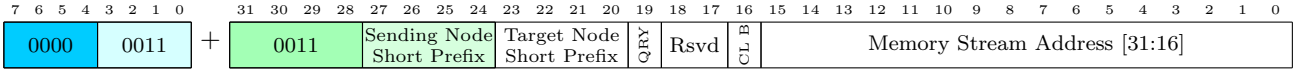
Member Node Set Memory Stream Address LSB



This message sets the lower halfword of the Memory Stream Write destination address for the target node specified in bits 23-20. If the `CL T` bit is high, the top halfword of the destination address is cleared (set to zero).

If the `QRY` (query) bit is set, the `CL B` and address are ignored. Instead, the node generates a **??** with the Queried Parameter field set to `0011`.

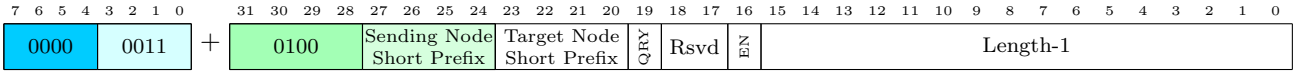
Member Node Set Memory Stream Address MSB



This message sets the upper halfword of the Memory Stream Write destination address for the target node specified in bits 23-20. If the `CL B` bit is high, the bottom halfword of the destination address is cleared (set to zero).

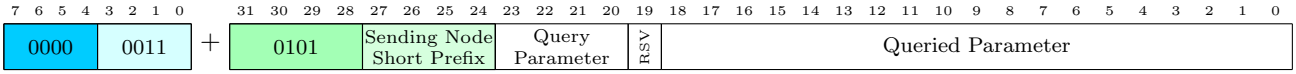
If the `QRY` (query) bit is set, the `CL B` and address are ignored. Instead, the node generates a **??** with the Queried Parameter field set to `0011`.

Member Node Set Memory Stream Length



TODO: Message

Member Node Query Memory Stream Response

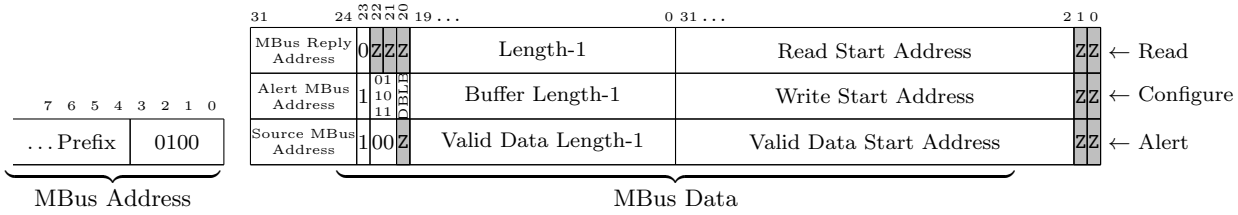


TODO: Message

Broadcast Channel 7: Data Channel 7 is used to send arbitrary data to every addressable entity on the bus. MBus places no further restrictions or structure on channel 7 messages. Channel 7 is intended for use by more flexible software nodes, though any node may listen or transmit on channel 7.

LU	Resolution	Range
00	1 word	1–64 words (4–256 B)
01	16 words	16–1K words (64–4K B)
10	256 words	256–16K words (1K–64K B)
11	4K words	4K–256K words (16K–1M B)

B.0.5 Memory Stream Read, Configure, or Alert



Three commands share the 0100 FU_ID. Bits 21–23 in the first word determine whether the command is a read, configure, or alert. A stream read generates a stream write. A stream configure is an isolated command with no response. A stream alert is generated in response to a stream write.

Read (Bit 23 == 0): The first word indicates the MBus address to reply to and the length of the requested read in words less one. The second word received is the address in memory to read from. The bottom two bits of the address field are reserved and **must** be transmitted as 0.

The response is sent immediately and the message is formatted exactly as the Memory Stream Write command, that is a series of sequential 32 bit data fields.

Overflow: 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 wraps and continues sending from address 0x00000000. *Tests:* **??**.

Interjection Semantics: If the reply is interjected, the transaction is aborted and is **not** retried.

Configure (Bit 23 == 1, Bits 22-21 in (01,10,11): Bits 22–21 specify which streaming channel is currently being configured. The first byte holds the MBus address that an Alert message will be sent to when this streaming channel's buffer is full. Bits 19–0 specify the maximum length of the buffer. The second word specifies the address in memory to begin writing to. Bit 20 DBLB specifies whether double-buffering mode is active. Without double-buffering, an Alert is generated once the buffer is full and the buffer cannot be written into again until another Configure is received. With double-buffering, an Alert is generated once halfway through the buffer and again at the end of the buffer. At the end of the buffer, the address resets to the beginning of the buffer and continues to accept writes.

Alert (Bit 23 == 1, Bits 22-21 == 00): The first byte holds the address of the node generating the Alert. The second word holds the start of the buffer in the local address space and the end of the first word specifies the length of the buffer that is valid.

asynchronous Send single word message to address.

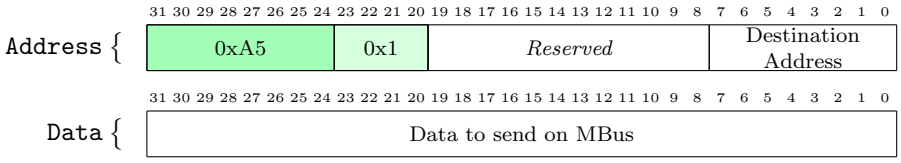
synchronous Retrieve received single-word message.

0xA51	W	??
0xA54	R	??

B.0.6 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.

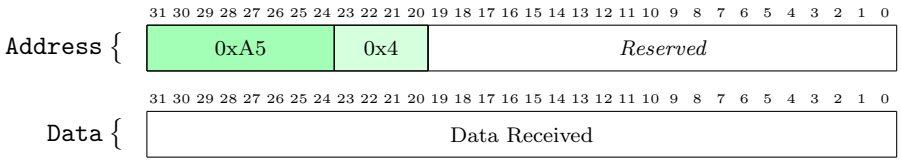
Bits Required	Purpose
Address	
8	MBus Memory Map Location
4	Command Specifier
8	Message Destination Address
Data	
32	Data to send on MBus



B.0.7 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 (8.4.1) 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



Thought from Return to Idle:

> *M3 Implementation Note:* (Referring to **??**) A broadcast sleep message cannot be considered a sleep until time 18 when the transmitter asserts that all the sent bits were desired to be sent. This leaves edges at time 20 and time 22 as the required two edges to power gate the layer.
>
> This could be complicated, however, if a node elects (for some reason) to send a response to the broadcast sleep message. The sleep controller will still have four edges (Arbitration, Priority Drive, Priority Latch, Begin Transmission) to wake its bus controller, but the timing between arming its `CLK_IN` fall detector and the mediator node pulling `CLK_IN` low in response could cause the sleep controller to miss wakeup.