1 Overview

Programmable hardware is increasingly deployed in large, physically distributed control systems. Hard real-time systems especially benefit from the determinism and low latency of purpose-built hardware. As reconfigurable hardware components replace traditional software-based systems, those hardware components must often communicate directly, over longer distances. While traditional protocols like CORBA and SOAP provide an excellent abstraction for software-to-software communication, they are a poor fit for hardware-to-hardware communication. Hardware components typically transfer information in read/write bus cycles as opposed to the procedure calling interfaces seen in software.

The Etherbone protocol takes an existing bus standard (Wishbone [?]) and extends this bus to run over the network. A concrete bus standard was chosen, because different bus protocols often differ enough that conversion reduces fidelity. Wishbone was chosen because it is an open standard, simple, and pipelining. The underlying transport protocol is left open, as Etherbone's requirements are easily met. This specification defines Etherbone for UDP and TCP.

Etherbone's key features are:

- Bus-cycle interface
- Deterministic latency
- Simple hardware implementation
- Compatible with software implementations
- Separate config/control address space
- Negotiable bus/address widths
- Pipelinable request processing

Etherbone leaves these issues to the lower transport layer:

- Exactly once delivery (TCP or reliable layer 2)
- Cut-through switching
- Authentication (physical access or TLS)
- Confidentiality

Etherbone was designed for the following use cases:

- High-precision system control
- Sensor data acquisition
- System diagnostics
- Remote debugging
- Distributed bus bridging

2 Architecture

Etherbone (EB) connects two Wishbone (WB) buses together as shown in Figure 1. The WB Intercon is a local bus, for example a crossbar interconnect.

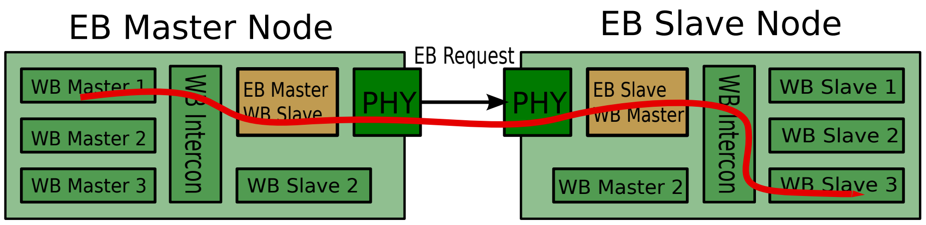


Figure 1: Etherbone system in hardware

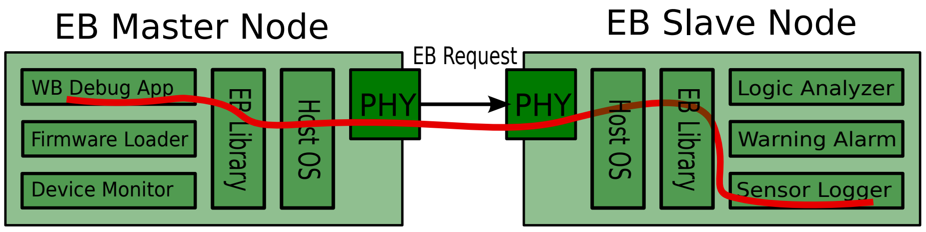


Figure 2: Etherbone system in software

When a WB master wishes to write to a remote slave, it writes to the EB bridge, which is a local WB slave. The bridge, acting as an EB master, translates the request into an EB frame (Section 5.1) and routes it to the receiving EB slave. That slave decodes the request and executes the write over Wishbone. Finally, the WB interconnect routes the write to the correct slave.

Either or both of the devices in Figure 1 may be replaced by software, as shown in Figure 2. In this scenario, the operating system buffers and sends Etherbone frames as requested by the Etherbone software library. Client applications may use this library for remote access to any slave attached to an Etherbone equipped wishbone bus. This facilitates such tasks as debugging, firmware updates, and monitoring from a dekstop system. Applications may also attach devices to a virtual wishbone bus in software, perhaps to capture bus cycles or trigger an alarm. These software devices may be mapped into the Wishbone bus of other Etherbone nodes, hardware or software.

2.1 Addressing

Slaves on a Wishbone bus have a mapped address range. Masters read and write to an address on the local bus and the Intercon routes the operation to the matching slave device. However, with the introduction of Etherbone, there are now multiple reachable Wishbone buses in the facility-wide system. To select the destination slave, additional address information is required.

When using the software interface, an application acquires a handle object for the remote bus. Reads and writes are then performed via the handle object, requiring only the WB bus address per operation. To acquire the handle object, the application supplies a device path identifying the target WB bus.

For a hardware implementation, the requests come from a local WB master, which can only provide a WB bus address. To determine the missing address information, an EB bridge must infer the destination WB bus based only on the local WB address requested. To achieve this, the EB bridge establishes a configurable mapping from local WB addresses to destination bus and target WB addresses.

For example, consider an EB bridge occupying address range 0x1000-0x3000 on the local bus. There is a remote WB bus available at udp/example.com/3434. We would like to access the address range 0x100-0x200 on that bus. Thus, we configure our EB bridge to map this range as 0x2000-0x2100 on the local bus. Now, when a WB write on our local bus to address 0x2050 is performed, the bridge transforms this into an EB write destined for the bus found at UDP address example.com:3434 and the WB address 0x150.

2.2 Pipelining

Unlike a local WB bus, where devices answer in a few clock cycles, a remote bus accessed via EB has a much high latency. For a 100MHz bus and a distance of only 20km the difference is 10ns to 100μs. For Internet-scale distances, the latency can easily rise to 100ms. Therefore, an application which only issues a new read/write operation when the previous operation completes will perform 10^4 to 10^7 times slower over EB than direct WB.

EB supports pipelining to overcome this significant performance bottleneck. Instead of issuing a single operation at a time, an application/device can issue new operations without waiting for the previous operation to complete. The results of the operations will arrive in the same order they were issued. Whenever new operations do not depend on still incomplete operations, this masks the performance lost to remote access.

As an example, considering two application using EB. The first application is a firmware writing tool that needs to write the firmware and confirm the firmware was written correctly. This problem can be readily pipelined; the operations have an order requirement (confirmation happens after write), but the choice of operation to issue does not depend on previous results. The firmware writer can issue a sequence of WWWW...RRRR... operations in the pipeline without waiting. Alternatively, it might also use the sequence WRWRWR... to confirm each word immediately after writing it. In both cases, the application can issue all of the operations without waiting. This would not be possible if the application were to iterate a remote function. Suppose the application wants to compute $f(f(f(...f(x)...)))$ using a remote WB slave to calculate function f. Here, the aplication writes x to the remote slave and reads back $y = f(x)$. Then the application writes y to the remote slave and reads back $z = f(y)$. The write pattern WRWRWR... is the same as the firmware loader, but here we can only pipeline a single write-read operation pair together. Until we have received the result $f(x)$, we cannot issue $f(y)$.

In Wishbone, several operations can be grouped into a single bus cycle. A particularly bad situation for Etherbone is when dependencies appear within a WB cycle. Generally, WB cycles acquire the device for use until cycle completion. On a local bus, any access pattern will work, as the operations will complete quickly and release the cycle line. However, when this happens with EB-sized latencies, a cycle might tie up a device for potentially unacceptable duration. Consider for example a WB cycle that reads from one address and writes the result to another address. Locally, there is no problem; the entire WB cycles executes in a few nanoseconds. However, when that same access pattern runs over the network, the slave device needs to wait for the read result to travel to the master and the final write to travel back. A very bad design.

Dealing with data dependencies within a cycle is a major complication addressed in the different implementation options described in Section 5.5.

2.3 Config Space

In addition to remote bus access, Etherbone also provides a configuration space. This config space is used to specify transmission parameters, recover bus error status codes, and match read results to the requests.

The config space is a complementary 16-bit wide memory map attached to every EB slave. EB requests can read/write to this configuration space in addition the the normal WB bus. The data port width of the config space always matches the data port width negotitated for EB. When a field is larger than the negotiated data port width, it is presented in bigendian order.

The config space is divided up into two regions: the register space and the implementation space. All addresses in the register space correspond to EB control registers, specified in this document. The register space spans addresses 0x0-0x7FFF. The implementation space is guaranteed to be free for whatever use a hardware/software implementation chooses. The implementation space spans 0x8000-0xFFFF.

Two important registers in the address space include the error status register, which reports WB error status codes, and the self-describing bus (SDB) pointer, which provides information about the slaves attached to a remote bus. The implementation space is often used by EB masters to pair requests to responses. Concretely, reads to an EB slave trigger a write back to the source EB master. Those writebacks are often sent to the implementation space where they can be handled by the EB core/code and invisible to the WB bus.

2.4 Bus Widths

In Wishbone, a bus may have a port width that is 8/16/32/64 bits wide. Thus, a master in one WB bus might write 32-bits at a time, while a slave in another WB bus expects 16-bits at a time. Etherbone makes no attempt to convert between differing port widths, because converting a 32-bit write into two 16-bit writes might change semantics.

However, Etherbone does negotiate which port widths are acceptable to both devices. This mostly affects software, which can meaningfully support access with different port widths. Hardware implementations will typically advertise and accept only one width. Once a data port width has been negotiated, say to 32-bits, the Wishbone select lines may still be used to write individual bytes (so long as the target slave device supports them).

Address width conversion, as oppposed to port width conversion, is relatively straight-forward. Address 0x0400 is that same as 0x00000400. Address spaces in WB are conceptually infinite, but in practice are constrained to a fixed width. If a 32-bit device is accessed by a 16-bit device, the 16-bit device can only see the low 16-bits of the larger device's address space.

Address width is negotiated by Etherbone simply to determine the amount of space to reserve for them in the wire format. A hardware implementation is free to only advertise address widths whose message alignment is convenient to process.

3 Considerations for Real-time Deployment

One of the key features of Etherbone is that it can used with hard realtime constraints and extremely low latency. In this scenario, Etherbone only forms a small part of the complete system. To meet deadlines, presumably most network components must make timing guarantees. The Etherbone software library, due to its dependency on the host operating system (OS), can only guarantee responsiveness on a real-time OS. Therefore, in a hard real-time system, the software library should only be used if the network stack makes timing guarantees.

In addition to any existing timing requirements, Etherbone requires attached slave devices to respond quickly. As detailed in Section 5.4.3, Etherbone slaves process individual requests as a stream while they arrive to avoid buffering delays. To support this behaviour, however, slaves must meet a deadline for their responses to be included in the reply.

Another important consideration for real-time deployment is cross-talk. If a slave is controlled by two masters, it is possible that one master will experience delays when the other has control. Similarly, if a device is both master and slave, it may not be able to service an incoming request while it is still performing its own outgoing request. For these reasons, we recommend designs with a single dedicated master and simple slave-only devices.

4 Programmer Guide

Etherbone is designed to transport the Wishbone bus over the network. In the Wishbone bus, every participant is either a master or a slave. Only masters may issue requests and only slaves may service them. An application may use Etherbone to implement either a bus master or slave. For both scenarios there is a hardware and software interface.

A request is either a read or a write and includes the target address. If the request is a write, it also includes the word to store. A response includes a success/failure status bit and (if a read) the word read. A master may issue

Type	Purpose
eb_status_t	An error status result from an Etherbone API call. Can be converted to text with eb_status(). Takes values from Table 4.
eb_socket_t	Top-level object handle used in the Etherbone library. An application must first create a socket to use any other library feature.
eb_flags_t	Options which control the type of eb_socket_t created.
eb_width_t	A bitmask which lists acceptable Wishbone bit widths. 8/16/32/64 are the possible alternatives.
eb_descriptor_t	An operating system specific socket handle. This can be extracted from an eb_socket_t to use with operating system event handling.
eb_address_t	A 64-bit wide unsigned integer type for Wishbone addresses. If the software application only advertises a 16-bit address bus, the unused high bits are required to be zero.
eb_data_t	A 64-bit wide unsigned integer type for Wishbone data. If the negotiated bus port width is less than this, the unused high bits are required to be zero.
eb_cycle_t	A collection of queued Wishbone read/write operations. Operations are executed in order with the target device held busy for the cycle duration. Operations will not begin execution until the cycle is closed.
eb_device_t	A handle to a remote Wishbone bus. Used to open a new eb_cycle_t.
eb_network_address_t	A hostname:port string used to identify a remote Etherbone bridge.
eb_handler_t	A virtual Wishbone device. These may be attached to an eb_socket_t and accessed remotely.
eb_user_data_t	An opaque pointer type. Whenever an Etherbone library permits a user callback function, a eb_user_data_t may be supplied that is provided to the callback upon completion. This is useful to record state associated with the operation needed upon completion.

Figure 3: Etherbone software library type definitions

multiple requests without waiting for a response. These requests are transmitted in ordered batches called cycles.

4.1 Software Interface

The Etherbone software library relies on the host operating system to implement TCP and UDP. Therefore, strict timing requirements cannot be assured and

Status Code	Purpose
EB_OK	The operation completed successfully.
EB_FAIL	The operating system reported an error.
EB_ADDRESS	Invalid address; too large for the negotiated bus.
EB_WIDTH	Invalid data; too large for the negotiated bus.
EB_OVERFLOW	Cycle length was too long for the chosen protocol. This only applies to UDP transport.
EB_BUSY	The object cannot be closed as it is in use by derived objects.

Figure 4: Etherbone result status codes

maximum throughput will be somewhat below the physical limit. Nevertheless, this interface to Etherbone is useful since the application can leverage a higher-level development enviroment.

4.1.1 Etherbone Sockets

All interaction with the Etherbone library begins with opening an Etherbone socket using `eb_socket_open`. The socket is used to send and receive Etherbone messages over TCP or UDP. In order for the library to process incoming messages, the socket must be regularly polled by the application. Be aware that the Etherbone library is not thread-safe. All access to a socket and any derived objects must be from a single-thread. Different threads may make use of distinct sockets safely, however.

There are three approaches a software Etherbone application can take to managing activity on the Etherbone socket:

1. Watch the Etherbone socket for activity in an external event loop, for example, a GUI's top-level main loop. To support this, the method `eb_socket_descriptor` allows access to the underlying operating system socket descriptor. The external framework can then watch this for readability, and call `eb_socket_poll` to process the pending messages.
2. Regularly call `eb_socket_poll` while doing calculations. This frees the application to fully utilize the CPU for calculations, but still handles Etherbone messages. The responsiveness and performance of Etherbone in this scenario will depend on how frequently `eb_socket_poll` is called. Every 1-4ms is a good target. If there are no calculations to perform, this approach wastes a lot of CPU time needlessly testing the socket.
3. Use `eb_socket_block` to halt the application until messages arrive. Then, run `eb_socket_poll` to process the events. Repeat until Etherbone is no longer of interest to the application. As opposed to polling, this lets the CPU idle when there is no pending traffic.

Figure 5 lists the prototypes of the relevant methods. See Figure 3 for an explanation of the types, and Figure 4 for the potential return codes. The socket

```
eb_status_t eb_socket_open(
    int           port,
    eb_flags_t    flags,
    eb_width_t    supported_addr_widths,
    eb_width_t    supported_port_widths,
    eb_socket_t*  result);

eb_status_t     eb_socket_close     (eb_socket_t socket);
eb_status_t     eb_socket_poll      (eb_socket_t socket);
int             eb_socket_block     (eb_socket_t socket,
                                     int timeout_us);
eb_descriptor_t eb_socket_descriptor(eb_socket_t socket);
```

Figure 5: Etherbone socket control methods

may be either TCP or UDP as controlled by the mutually exclusive eb_flags_t options EB_UDP_MODE and EB_TCP_MODE. A valid port width argument is the bitwise OR of EB_DATA{8,16,32,64}, or EB_DATAX for all of them. Similarly, EB_ADDR{8,16,32,64,X} defines the acceptable address widths. The header files provide detailed pre- and post-conditions for using the methods.

4.1.2 Master Mode

An Etherbone application can become a master on a remote Wishbone bus. To begin this mode of operation, the application first opens the remote bus with `eb_device_open`. This initiates negotiation of the bus address and data widths, resulting in a wire format to be used for Etherbone cycles. The relevant methods appear in Figure 6.

Thereafter, the Etherbone application creates bus cycles. Each cycle performs a sequence of read or write operations [1]. Until the cycle is closed, it is not queued to be sent. Thus, data dependencies (Section 2.2) within a cycle are prevented, greatly improving the responsiveness of Etherbone.

Because network protocols perform most efficiently with larger transfers, the Etherbone library delays transmission of a cycle until either the local buffer is full or the the buffer is flushed with `eb_device_flush`. The methods involved are listed in Figure 7,

Putting this all together, to write to a remote register an Etherbone application needs to take these steps:

1. Open an Etherbone socket (`eb_socket_open`)
2. Open an remote device's bus (`eb_device_open`)
3. Begin a new Wishbone cycle (`eb_cycle_open`)
4. Enqueue a write operation (`eb_cycle_write`)
5. Mark the end of the cycle (`eb_cycle_close`)

[1]For UDP sockets, only 150 operations may be queued per cycle.

```
eb_status_t eb_device_open(
    eb_socket_t          socket,
    eb_network_address_t ip_port,
    int                  attempts,
    eb_device_t*         result);

eb_status_t eb_device_close        (eb_device_t device);
eb_socket_t eb_device_socket       (eb_device_t device);
eb_width_t  eb_device_port_width   (eb_device_t device);
eb_width_t  eb_device_address_width(eb_device_t device);
```

Figure 6: Etherbone remote bus access methods

```
typedef void (*eb_cycle_callback_t)
   (eb_user_data_t, eb_status_t, eb_bool_t*);

eb_cycle_t eb_cycle_open_read(
    eb_device_t          device,
    eb_user_data_t       user,
    eb_cycle_callback_t  cb);

void eb_cycle_close(eb_cycle_t cycle);
void eb_cycle_read (eb_cycle_t, eb_address_t, eb_data_t* data);
void eb_cycle_write(eb_cycle_t, eb_address_t, eb_data_t  data);

void eb_device_flush(eb_device_t socket);
```

Figure 7: Etherbone remote cycle methods

6. Flush the buffers out the socket (`eb_device_flush`)
7. Close the device and socket or begin a new cycle.

After a cycle has been executed remotely, the application-supplied callback will be invoked indicating that the reads have completed and providing error status for each operation.

Due to the packet format of Etherbone (Section 5.1), be aware that writing sequentially is more bandwidth efficient than random access.

4.1.3 Slave Mode

An Etherbone application can register virtual devices on a simulated Wishbone bus. These devices may then be remotely accessed by Etherbone masters. There is one virtual bus per Etherbone socket. The EB address is the application system's hostname and the port provided on socket creation.

For maximum interoperability, software slave devices should support access at any data width; this is not much of a burden when compared with hardware.

```
typedef struct eb_handler {
  eb_address_t base;
  eb_address_t mask;

  eb_user_data_t data;

  /* Values for Wishbone device autodiscovery */
  uint16 wbd_version;
  uint32_t vendor;
  uint32_t device;
  uint32_t wbd_flags;
  uint32_t hdl_class;
  uint32_t hdl_version;
  uint32_t hdl_date; /* e.g: 0x20111225 -- christmas 2011 */

  eb_data_t (*read) (eb_user_data_t, eb_address_t, eb_width_t);
  void      (*write)(eb_user_data_t, eb_address_t, eb_width_t,
                     eb_data_t);
} *eb_handler_t;

eb_status_t eb_socket_attach(eb_socket_t, eb_handler_t handler);
eb_status_t eb_socket_detach(eb_socket_t, eb_address_t address);
```

Figure 8: Etherbone virtual device methods

Slaves on a socket must all share the same supported address and port widths, as they are all on the same virtual Wishbone bus.

A virtual device claims a base address and mask, in the virtual socket bus. This address range must not overlap any other virtual slave devices attached to this socket. Address decoding and bus arbitration are handled by the library internally.

Each virtual device provides a read and write callback, as shown in Figure 8. These callbacks must return immediately and may not generate a bus error. Accesses to unmapped virtual addresses will always produce a bus error.

4.2 Hardware

To be written by Matthias.

5 Protocol and Implementation

This section details the Etherbone protocol. All conformant devices must adhere to the message format outlined in Section 5.1. Optional design considerations are explicitly indicated in the text. Every other behaviour described in this section is mandatory for a conforming Etherbone hardware or software implementation.

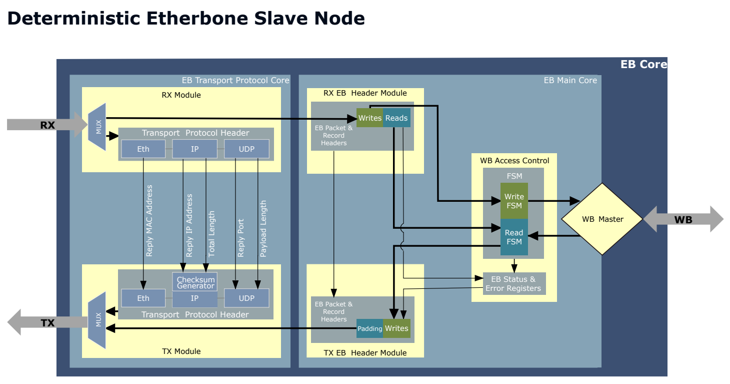
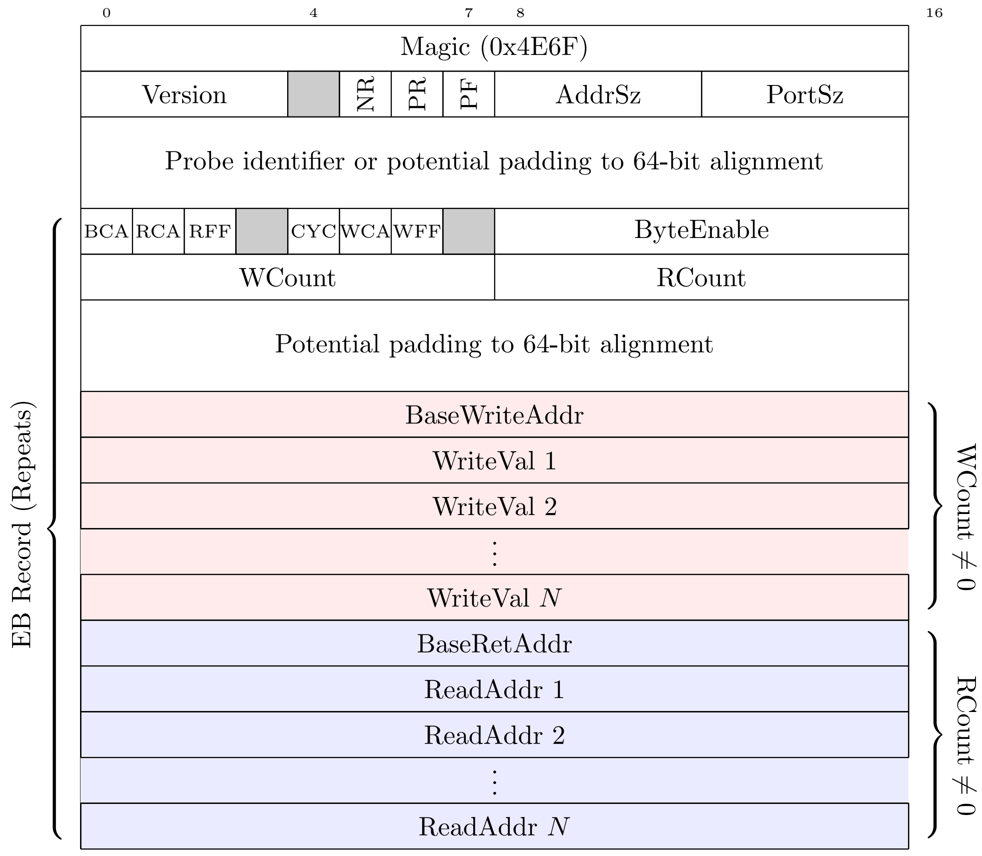


Figure 9: Etherbone hardware

All reserved fields (indicated in grey) must be cleared to 0. All numeric values have bigendian format and this documentation is written in most significant bit 0 (MSB0) format.

5.1 Message Format



5.1.1 Etherbone Header

An Etherbone message contains an Etherbone header followed by a sequence of records. Records are processed in the order they appear in the message.

The packet is aligned to the maximum of 16 bits, the negotiated address width, or the negotiated data port width. For Etherbone version 1, this is either 16, 32 or 64 bits. If the packet has 64-bit alignment, the padding fields must be included.

Magic As Etherbone is layered atop UDP or TCP, stray messages might find their way to Etherbone endpoints. In order to indentify the Etherbone protocol, every valid Etherbone message includes the value 0x4E6F as a prefix. Any packet missing this magic protocol identifier must be dropped.

Version This document describes Etherbone version 1.

Future Etherbone revisions must retain support for obsolete Etherbone messages formats. Etherbone responses must use the same version as the request which triggered the response.

If an Etherbone device receives a request for an unsupported version, it should ignore the request unless PF is set. When PF is set, the largest of the requested version and the supported version should be returned.

PF The Probe-Flag (PF) is used to negotiate compatible bus widths and version. Probe messages include the 4 byte probe identifier field and thus are always exactly 8 bytes long. The highest Etherbone version supported appears in Version. The contents of AddrSz and PortSz describe the data port and address widths supported by the probing device. Upon receipt of a message with PF set, an Etherbone device should respond with a Probe-Response.

PR The Probe-Response (PR) is used to negotiate compatible bus width and version. PR messages echo back the probe identifier from the initiating PF message and are thus always exactly 8 bytes long. The highest Etherbone version supported appears in Version. The contents of AddrSz and PortSz describe the data port and address widths supported by the probed device.

NR The No-Reads flag (NR) indicates that the entire message contains no read operations. An Etherbone device must leave this field 0 if the message includes reads. If the message is a response containing no reads, the field must be set to 1. If the message is a request containing no reads, the field may be set 1.

The purpose of this flag is to tell a streaming Etherbone slave that it need not send a response message. Streaming devices begin transmitting before they have fully received the incoming message, so they cannot determine ”no reads” at the time transmission begins.

AddrSz In principle, Wishbone buses do not have a fixed address width. However, addresses are packed into fixed width fields in Etherbone. Etherbone negotiates the width of all address fields. An Etherbone device lists all the address widths it supports in AddrSz. The values below are bitwise ORed together to indicate supported widths.

AddrSz	Address bits
1	8
2	16
4	32
8	64

PortSz The Wishbone standard permits interconnection of slaves and masters with bus data widths of 8/16/32/64. Masters must use the correct data bus width when communicating with slaves. Etherbone negotiates the data port width used in requests. An Etherbone device lists all the port widths it supports in PortSz. The values below are bitwise ORed together to indiciate supported widths.

PortSz	Data bus width
0	8
2	16
4	32
8	64

5.1.2 Record Header

Each record in an Etherbone stream includes a header. If the packet alignment exceeds 16-bits, the header is followed by zeros padding to the alignment size.

RCA If the addresses read in this record should be taken from the config address space, the RCA flag is set. Otherwise values will be read from the Wishbone bus.

BCA If the BaseRetAddr is in the config space of the originating EB device, the BCA flag is set. The value of this flag is copied to the matching WCA field.

RFF If the results of Wishbone reads should be written back to a FIFO register located at BaseRetAddr, the RF flag is set. Otherwise read values will be written sequentially starting at the BaseRetAddr.

CYC If the contents of this record are the last of a cycle, the CYC flag is set. A receiving EB device should hold the cycle line high till it encounters this flag.

WCA If the values written in this record should be written to a the config address space, the WCA flag is set. Otherwise values will be written to the Wishbone bus.

WFF If the values written in this record should be written to a FIFO register, the WF flag is set. Otherwise values will be written sequentially starting at BaseWriteAddr.

RCount This Wishbone record includes RCount reads. The message will include an equal number of ReadAddr fields.

ByteEnable All operations in this record use these byte enable bits on the Wishbone bus. For 64-bit data port width, all bits are used. For 8/16/32-bit widths, the rightmost (least significant) bits are used. The least significant bit 15 matches the least significant byte of WriteVal.

WCount This Wishbone record includes WCount writes. The message will include an equal number of WriteVal fields.

5.1.3 Read Data Section

Read fields are processed in order.

BaseRetAddr This field is present only if RCount > 0. If the AddrSz is shorter than the packet alignment, the address is big-endian zero extended. When it exists it indicates the address on the origin Etherbone endpoint to which read results should written.

ReadAddr For each ReadAddr, a read operation to that address will be executed. Depending on the value of RCA, the address is either a config space or Wishbone bus read. If the AddrSz is shorter than the packet alignment, the address is big-endian zero extended. Results are written back to BaseRetAddr on the origin endpoint.

5.1.4 Write Data Section

Write fields are processed in order.

BaseWriteAddr This field is present only if WCount > 0. If the AddrSz is shorter than the packet alignment, the address is big-endian zero extended. When it exists it indicates the address on the target Etherbone endpoint to which values should written. If WCA is set, writes go to the config space, otherwise the address is on the Wishbone bus.

WriteVal For each WriteVal, a write operation will be executed. If the PortSz is shorter than the packet alignment, the value is big-endian zero extended. The destination address is computed from BaseWriteAddr and WFF.

5.2 Config Registers

Config registers are always 64-bits wide. Access to the config registers must always be aligned; a 32-bit read to address 2 has undefined result. Access to the config registers must always use the full negotiated port width; all byte enable bits must be set. So far, there are only two config registers defined:

Register	Purpose
0	Error status. This shift register records the error status for completed Wishbone operations. If the last operation was successful, the low bit of this register is zero. Therefore, after an Etherbone master issues a cycle containing 13 operations, it can read this register to determine the error status using the low 13-bits of the register. Config space read/writes leave this register unmodified.
8	Address of the Wishbone autodiscovery header. Every Etherbone device must include somewhere in its config address space a Wishbone autodiscovery structure. This structure describes all devices attached to the local Wishbone bus.

5.3 UDP and TCP streams

The Etherbone protocol is embedded directly in both UDP and TCP. In both cases, the protocol format and handling remains the same. The Etherbone protocol is designed to be processed as a stream. For UDP datagrams that stream is simply packed into a single message. For TCP, the stream consumes the entire TCP connection, probably split into many TCP segments.

With UDP operation, each UDP message includes a fresh Etherbone header. The packet contents can be considered as a (short) Etherbone stream whose format adheres to Section 5.1. A master should end the cycle by setting CYC in the last record. By design, an Etherbone response can never be longer than the Etherbone request which triggered it. Therefore, whenever a UDP request arrives, the UDP response will fit in the same sized frame. Etherbone does not add reliability to UDP. If reliable operation over UDP is required, link layer FEC should be used.

For TCP operation, the entire TCP stream is a single Etherbone stream; there is only one Etherbone header at the very beginning of the stream. TCP adds reliability and congestion control to Etherbone, but at the cost of buffering delay and (potentially) retransmission.

For real-time reliable hardware, UDP is the appropriate choice. It is faster, deterministic, and cheaper to implement in hardware. Whenever software requires access over an unreliable link, like the global Internet, TCP is the better choice. An Etherbone TCP-UDP gateway can be used to bridge a reliable internal network to the wider internet.

5.4 Server / Slave Operation

The Etherbone format was carefully designed to make it a stream process. It is possible to process an incoming UDP datagram and begin sending an outgoing response This approach is described in Section 5.4.3.

5.4.1 Header processing

Any stream with invalid Magic is ignored in its entirety. Otherwise the Magic is copied to the outgoing stream.

The maximum of the received Version and locally supported version is taken as the version of the following Etherbone payload. This allows a negotiation scheme whereby two peers exchange Etherbone headers and can then deduce the correct wire format.

Similarly, the received AddrSz and PortSz is bitwise ANDed with the locally supported widths to determine overlap. The largest version supported in common is used. If there is no commonly supported width, the connection is terminated.

If the PF flag is set, then the response must set the PR flag. Both flags indicate the presence of a 32-bit probe identifier in the stream. When responding to a probe, the response must copy this identifier.

5.4.2 Record processing

A response record header is created from a request record header by the following transformation:

- 0 → RCA
- 0 → BCA
- 0 → RFF
- CYC → CYC
- BCA → WCA
- RFF → WFF
- 0 → RCount
- RCount → WCount
- ByteEnable → ByteEnable

If RCount is non-zero, the BaseRetAddr is copied from the input stream to the output stream where it will fill the BaseWriteAddr slot. Each ReadAddr is then read from either the config space or Wishbone bus according to the RCA flag. The results are written to the output stream.

If WCount is non-zero, the BaseRetAddr is stored to a variable x. For each WriteVal, a write is issued to address x on either the Wishbone bus or config space depending on the WCA flag. If WFF is set, x is incremented by PortSz after each operation.

While processing writes, an Etherbone slave has two options for the output stream. It can either write nothing, saving bandwidth, or it can write WCount+1 zeros. The advantage to filling in zeros is that it makes the outgoing stream exactly the same length as the incoming stream.

After completion of each Wishbone read or write operation, the EB slave shifts the error status into config register 0 (Section 5.2). This can be read by the EB master to determine the error status of the operation.

5.4.3 Streaming

To achieve extremely low latency, a hardware Etherbone slave can operate in streaming mode. Here, each read/write operation is pipelined to the Wishbone bus while the packet is still being received. As the Wishbone acknowledgements arrive, the outgoing stream is filled in normally.

In order to stream an outgoing UDP frame, the slave must implement a buffer large enough to fit an IP+UDP header. Once the slave has determined that it will answer a request (after processing the Etherbone header), it fills the IP+UDP headers with fields appropriate to respond to the origin device. The UDP checksum is set to 0 as the UDP payload is not available for checksum in time for streaming mode; this is allowed by RFC 768. The outgoing IP length is set to the same as the incoming length; this is why Etherbone supports zero padding for write operations.

Streaming out this initial UDP+IP header buys the local bus some latency for responding to requests. However, the local Wishbone bus must process requests faster than the physical connection Etherbone uses to send. Otherwise, the out-going Etherbone stream would starve and create a corrupt packet. Real-time implementations must, therefore, ensure every slave on their Wishbone bus has sufficient response-time.

One wrinkle that must be dealt with is Wishbone cycle termination. There are three problems: stalling between cycles, status register stalls, incomplete cycles. They will be discussed in turn.

Unfortunately, a Wishbone cycle cannot end until all the acknowledgements have completed. Therefore, before processing the next cycle in the stream, the Etherbone slave must stall until the previous cycle completes. Each such stall slowly eats into the latency margin won by the IP+UDP header. The only way to ensure these stalls do not starve the response stream is to guarantee that every operation is acknowledged within the time it takes to stream three words minus two cycles. Since the record header and address can be queued in one cycle each, every record wins two words of latency minus two cycles. Since every cycle includes at least one record and one operation, this means the bus can always keep up with transmission if the pipeline depth does not exceed three words delay. To make this concrete, suppose the Wishbone bus is 32-bit wide 125MHz and the output is Gigabit ethernet. Now, every attached Wishbone slave must guarantee acknowledgement within 10 cycles (3*32*125MHz/1GHz-2).

The next problem is that the error status register depends on Wishbone error status codes. Those codes only arrive with the operation completion. Thus, whenever a record reads the config space, the EB slave must stall until all outstanding Etherbone operations complete. Fortunately, the same restriction needed for cycle termination stalls is sufficient to also cover this case. Every transition from Wishbone bus access to config space access costs at least two words for the new record header.

Finally, there is the sticky issue about how to handle a stream that never ends the cycle. There are two options: honour this request or ignore it. If the device chooses to honour the request, it will block the local Wishbone bus until the next stream arrives. If it does not honour the requset, an operation that should have been atomic might be broken. Notice that a software Etherbone master / client will never issue a request of this form. Unfortunately, transparent bridging (Section 5.5.3) always generates requests like this. Ultimately, further operations continuing a cycle from a previous stream is a pathological case caused by a poorly designed WB master with an in-cycle data dependency (Section 2.2). The chosen behaviour is left open as an implementation decision, but it is probably wisest to implicitly terminate cycles on stream end.

5.5 Client / Master Operation

An Etherbone master issues requests to remote Wishbone buses. It might be a software client, a hardware EB client, or a bridge device that maps a remote bus locally. In any case, the task is fairly straight-forward: transform read/write operations into Etherbone messages.

One important design consideration is how to handle read results. The config space reserves the range 0x8000-0xFFFF for implementation use. Probably the best implementation approach is to use this space as the BaseRetAddr in read requests. The resulting write-back can then be matched to the triggering read.

5.5.1 Software Client

A software client is by far the simplest. A small internal table records the callback information needed for each outstanding cycle. The BaseRetAddr is used as an index into this table.

Read/write operations are enqueued on a per cycle basis. Once the cycle is closed, the library looks to see if the writes can be compressed into a FIFO or sequential write stream. Otherwise it just streams the operations out as a sequence of records. Every 64 operations, it injects a config register read to recover error status codes.

5.5.2 Hardware Client

A hardware client is not much more sophisticated than a software client. We assume that this client is generating the requests itself, not translating them from a local bus.

Presumably, it's requests have a regular format and it doesn't even need the sophistication of a software client's pattern matching for FIFO/sequential access.

For demanding real-time applications, a dedicated hardware Etherbone master is the best approach. Bridging can achieve similar latency, but only a hardware client can fully utilize available bandwidth.

5.5.3 Bridge Mode

Unfortunately, a Wishbone cycle cannot end until all the acknowledgements are received. Wrose, the Etherbone bridge cannot issue acknowledgements until it receives Etherbone responses. Thus, Wishbone cycles cannot complete faster than a network round-trip. This completely kills the throughput of bridged bus. Despite the poor throughput, reasonable latency can still be attained.

Keeping this in mind, we can safely assume that any high throughput Etherbone master will use a dedicated hardware client and not be built on a bridge. Many sticky implementation issues below can be resolved by trading bandwidth for simplicity. Given the inherent limitations to bridging, this seems a reasonable trade-off.

A bridge should start streaming out a fixed length UDP/IP Etherbone request as soon as a Wishbone cycle begins. Without buffering, an unknown access pattern cannot be compressed. Therefore, a bridge simply translates each operation into a separate Etherbone record. Whenever there is no pending operation, the bridge just writes zeros (an empty Etherbone record). If operations come faster than space is available in the stream, stall the Wishbone master until space is available. When the currently streamed message runs out of space, stall the master until the response has arrived. As most Wishbone cycles are short, this is a pragmatic way to avoid the huge buffers needed to fixup reponse message mis-ordering.