



HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.co.uk
<http://www.hunteng.co.uk>
<http://www.hunt-dsp.com>



Heron Serial Bus (HSB)

Specification

Document version 1.5

P. Warnes 13/03/03

COPYRIGHT

This documentation and the product it is supplied with are Copyright HUNT ENGINEERING 1999. All rights reserved. HUNT ENGINEERING maintains a policy of continual product development and hence reserves the right to change product specification without prior warning.

WARRANTIES LIABILITY and INDEMNITIES

HUNT ENGINEERING warrants the hardware to be free from defects in the material and workmanship for 12 months from the date of purchase. Product returned under the terms of the warranty must be returned carriage paid to the main offices of HUNT ENGINEERING situated at BRENT KNOLL Somerset UK, the product will be repaired or replaced at the discretion of HUNT ENGINEERING.

Exclusions - If HUNT ENGINEERING decides that there is any evidence of electrical or mechanical abuse to the hardware, then the customer shall have no recourse to HUNT ENGINEERING or its agents. In such circumstances HUNT ENGINEERING may at its discretion offer to repair the hardware and charge for that repair.

Limitations of Liability - HUNT ENGINEERING makes no warranty as to the fitness of the product for any particular purpose. In no event shall HUNT ENGINEERING'S liability related to the product exceed the purchase fee actually paid by you for the product. Neither HUNT ENGINEERING nor its suppliers shall in any event be liable for any indirect, consequential or financial damages caused by the delivery, use or performance of this product.

Because some states do not allow the exclusion or limitation of incidental or consequential damages or limitation on how long an implied warranty lasts, the above limitations may not apply to you.

TECHNICAL SUPPORT

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

Change History

- 1.0 First Written
- 1.1 Added HEART expansion modules
- 1.2 Corrected C6000 module query response definition
- 1.3 Added new "special options request" message for FPGA modules. Implemented only by HERON-FPGA4 and later.
- 1.4 Added Server/Loader run type request.
- 1.5 Added the Compressed FPGA download commands

TABLE OF CONTENTS

WHAT IS HSB?	5
WHY DO WE NEED HSB?	6
WHAT DOES IT ALLOW US TO DO?.....	7
IS IT A STANDARD?.....	8
HSB ELECTRICAL DEFINITION	9
LOW LEVEL PROTOCOL	10
RESULTING CAPABILITY	12
SUPPORTED I2C CYCLE TYPES	13
HSB ADDRESS DEFINITION.....	14
HSB MESSAGE TYPES	15
USING HERON-API FUNCTIONS TO SEND AN HSB MESSAGE.....	16
USING HOST-API FUNCTIONS TO SEND AN HSB MESSAGE	17
APPENDIX 1 – MESSAGE DEFINITIONS BY MODULE CLASS.....	19
C6000 PROCESSOR MODULES	19
<i>Module Query and response</i>	<i>19</i>
<i>Server/Loader run query.....</i>	<i>19</i>
FPGA MODULES/HERON_IO MODULES THAT HAVE AN FPGA	20
<i>Module Query and response.....</i>	<i>20</i>
<i>Configure FPGA.....</i>	<i>21</i>
<i>Configure FPGA with compressed format.....</i>	<i>21</i>
<i>User Read and Write Access.....</i>	<i>22</i>
<i>Special Options and response.....</i>	<i>22</i>
I/O ONLY MODULES	23
<i>Module Query.....</i>	<i>23</i>
<i>User Read and Write Access.....</i>	<i>23</i>
HEART FPGAS.....	25
<i>HEART Configuration</i>	<i>25</i>
HEART EXPANSION MODULES.....	25
<i>Module Query.....</i>	<i>25</i>
<i>Make connection.....</i>	<i>26</i>

HSB is a serial bus that is part of the HERON Specification. It has pins allocated on the module pinout, and uses a protocol that has been defined by HUNT ENGINEERING. This document defines the electrical signals and the protocols used.

A HERON system is designed for real time operation, using connections based upon FIFOs that guarantee no arbitration delay, and the (maximum) bandwidth available. Some system functions do not require these features, and it would be a waste of system resources to use a FIFO connection for them.

What does it allow us to do?

The HSB is a multi-master arbitrated bus that allows any node to address any other node. It uses only two signals (and needs a common ground) to perform arbitration, addressing and passing of data. It caters for devices that have different maximum bit rates, and for devices that are interrupt driven for servicing the commands.

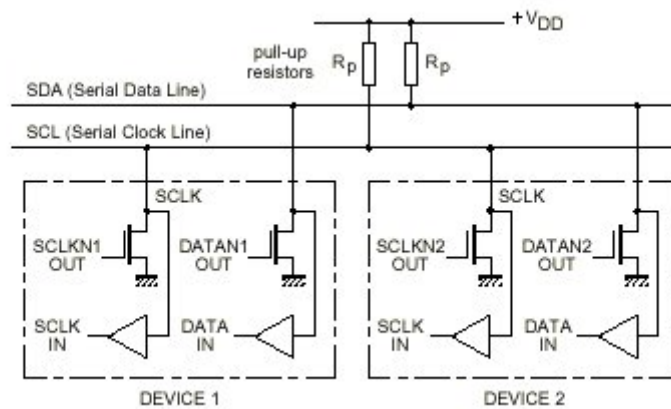
HSB has been based on the electrical definitions of the I2C bus defined by Philips. It has departed from the I2C in terms of protocols, and bit rates to the extent that it cannot be assumed that an I2C device can be connected and expected to work. In some cases low level interface devices that are designed for I2C can be used, for example the Philips PCF8584 was used on the HEPC8 HERON carrier.

The trend within HUNT ENGINEERING is to implement the HSB interface within an FPGA component allowing total control over the functions implemented.

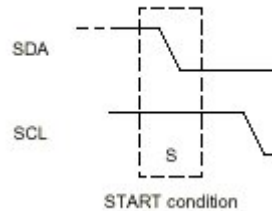
The HSB is made up of two signals, a clock (SCLK) and a data (SDA) signal. Both signals are open drain, with a pull up which means the inactive state (no-one driving the bus) is to have both signals high. This is defined as the bus free state.

The pull up should be 10K, and to +3.3V although some early carriers were 5V based, and would connect the 10K to +5V.

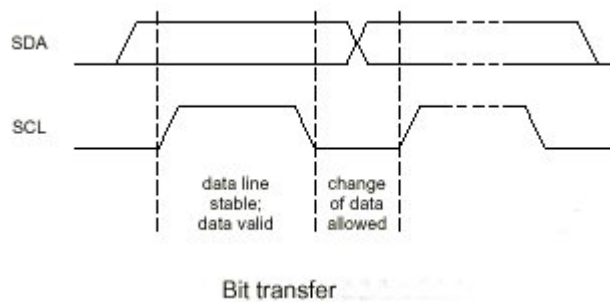
The state of the bus must be readable during transmission so that arbitration can take place.



A message is begun with the start condition, which is a falling edge on the data line, while the clock line remains High.



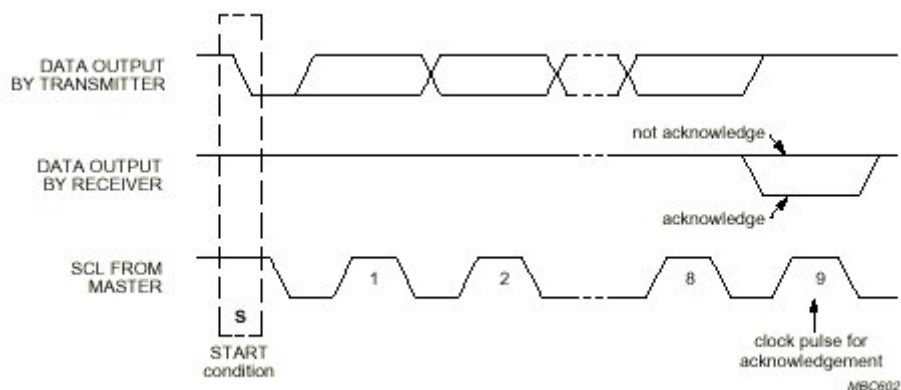
After that, it is not allowed to change the state of the data line while the clock line is high, and data bits are transmitted by the receiver taking the value of the data line on each rising



edge of the clock line.

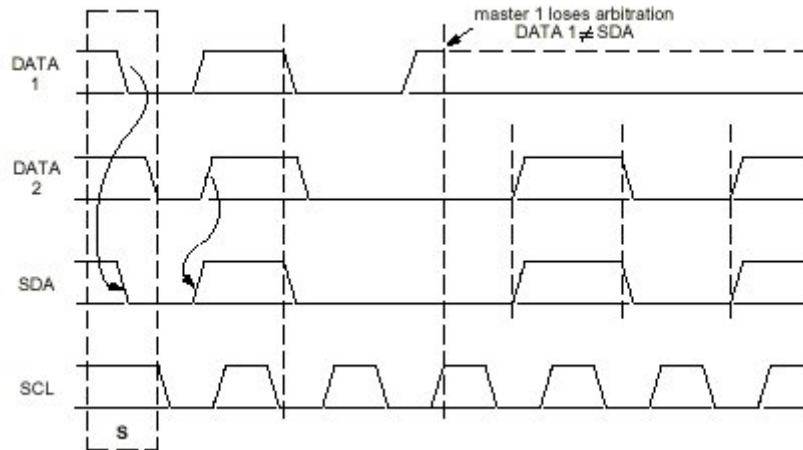
Because the signals are open drain, the receiver can pull the clock low for the length of time it takes to “process” that bit. This means that each rising edge will occur only after the sender and the receiver are ready.

Data is sent in bytes (groups of 8 bits), with a ninth clock used to “acknowledge” that the receiver has received the byte. For the first 8 bits the sender drives the data, and on the ninth, the receiver drives the data low to acknowledge the byte. If it is not driven by the receiver, the pull up on the open drain signal takes the acknowledge bit high creating a “not acknowledged” result.



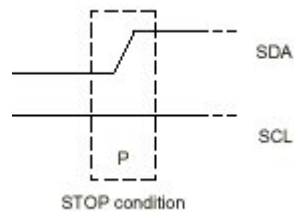
In I2C the first byte forms an address, with the bottom bit indicating a read or a write cycle. HSB supports only Write cycles, (read is accomplished by sending a read request message using a write cycle) so the LSB is always zero. HSB defines the remaining seven bits to be made up of three bits of module ID and four bits of board ID.

If several nodes detect the bus free at the same time and proceed to form an address, the I2C arbitration process will give a winner and a loser. This is detected by comparing each bit sent with the state of the SDA line after a time delay that allows for bus settling. If the actual state of the SDA line differs from the state that a node is driving, then it has lost arbitration, and must immediately stop transmission.



The winner of the arbitration will succeed in its transmission without having to re-transmit, the HSB logic should inform the loser so that re-transmission can start when the bus is free again. This mechanism means that if two nodes attempt to transmit to different targets, there will only be one winner. The winner is defined by the address, as the loser will lose on the first bit it fails to drive (so is high) and is actually driven low by the winner. If the addresses are the same, then arbitration will continue until a data bit is different, and the same arbitration technique will determine who is the winner. This is not entirely a “fair” scheme, but the use of the HSB is such that traffic is not heavy, and arbitration clashes are not expected to occur frequently.

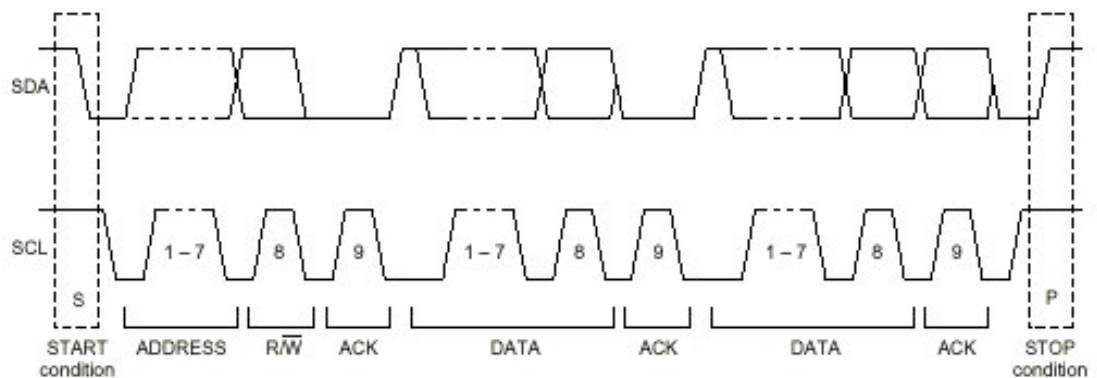
Once a message is started with the start condition and the target address, data transmission continues until the stop condition is presented by the master. The stop condition is a rising edge on the data line while the clock line is high.



The use of this I2C like protocol allows messages to be arbitrary lengths.

It also means that the clock rates of the sender and receiver do not need to be the same. In a system of devices capable of differing speeds the speed of the slowest device in a transaction sets the speed.

The use of 7 bits of address defines the remaining seven bits to be made up of three bits of module ID allowing up to eight devices per board, and four bits of board ID, allowing up to 16 boards per system.



Supported I2C cycle types

The I2C bus supports master and slave accesses, each of which can be a read or a write. The HSB supports only Master Write type accesses. This means that the only sequence of events is the bus free state is detected, so a START condition is generated and an address is presented on the HSB. Then the HSB protocol is sent, terminated by the STOP condition.

If a response is required, the original receiver will then arbitrate for the bus and send the reply as a separate Master Write message.

Each module is addressed based on its slot number on a board and the setting of the board number switch on the carrier board.

The HSB basic address is 7 bits, which in a HUNT ENGINEERING system is formed as follows:-

Bit 0-2 node number

0=reserved

1,2,3,4 HERON slot number

5=host bus's HSB node

6=inter-board connection node

7=HEART FPGAs for routing configuration.

Bit 3-6 board number, as set by the board ID switch

HUNT ENGINEERING have defined the general structure of an HSB message to be :-

Start Condition	Target Address	Message Type	Reply address (optional depending on message type)	N bytes of data as defined by message type	Stop Condition
-----------------	----------------	--------------	--	--	----------------

The start and stop conditions are mandatory along with the Target address that specifies which node the message is for.

The message type is a HUNT defined byte. At this time the following message types are defined :-

Message type	description
01	Module Query
02	Module Query reply
03	FPGA configuration message
05	FPGA configuration success
06	FPGA configuration fail
07	HEART configuration message
08	User Write message
09	User read request
10	reply from user read request
11	Special Options request
12	reply from Special Options request
13	FPGA compressed configuration message
175	Server/Loader run request (0xaf)

Other message types are reserved. If you need to use a message type in your system and wish to ensure that this will not clash with messages used by HUNT ENGINEERING, then tell us what you want to use the message for. We may allocate a message type for that purpose, guaranteeing that it will not be used by HUNT other than for that purpose.

Using HERON-API functions to send an HSB message

HERON-API offers two methods of using the HSB from a C6000 DSP processor. There is a “Quick method” that can be used to form simple messages, and a more involved method that allows you to form messages in any format.

As with all HERON-API based accesses you must first “open” the device using the `HeronHsbOpen()` function and obtain a handle.

The Quick method uses the functions `HeronHsbSendMessage` and `HeronHsbReceiveMessage`.

Calling the Send message function as follows :-

```
status = HeronHsbSendMessage(hsb,USER_READ,board,targetslot,send_data,2);
```

uses the handle “hsb” obtained from the open function, to send a user Read message as below

Start Condition	Target Address formed from Board and target slot	Message type = USER_READ	Reply address formed from the data held in the handle “hsb”	2 data bytes taken from send_data[0] and send_data[1]	Stop Condition
-----------------	--	--------------------------	---	---	----------------

Calling the function receive message as follows :-

```
status = HeronHsbReceiveMessage(hsb, &message_type, board, targetslot, reply, buffer_size, &done);
```

Will use the device “hsb” (obtained from the open function) to receive a message. The function will not complete until a message has been received. If a message is received from the “expected” master, i.e. that specified by forming an address out of the “board” and “targetslot” parameters, the message type variable will be filled in, along with the values `reply[0]`, `reply[1]` etc. The number of valid data bytes is shown by the variable “done”, which will not be allowed to exceed the value `buffer_size`, even if more data is actually returned in the HSB message. If the message is not from the “expected” master and error will be returned.

For more details on these functions please refer to the HERON-API user manual.

The more involved method breaks the message down into start, data and stop phases. Typically this is used for messages that have long data sections where it is not sensible to copy all of the data into a local array.

Again for details of these functions refer to the HERON-API user manual.

Using Host-API functions to send an HSB message

As with the HERON-API, to write or read the HERON serial bus using the Host API, first open it with an HeOpen call, using device name “hsb”. This allows the Host PC to use the features of HSB.

The Host API offers 3 types of access methods, called “level3”, ‘level 2’ and ‘level 1’. The highest level is easiest to use, it consists only of two functions: send a message and receive a message. If the message is long or if the message should ideally be sent or received in parts, use ‘level 2’ functions. Each ‘level 3’ function can be functionally split up in 3 ‘level 2’ functions. The ‘level 3’ send message function splits into a ‘start send message’, a ‘send message data’, and an ‘end of send message’ function in ‘level 2’.

The ‘level 1’ functions are low-level functions and you are advised not to use them.

The Level 3 method uses the functions HeHSBSendMessage and HeHSBReceiveMessage.

Calling the Send message function as follows:-

```
Status = HeHSBSendMessage(hsbDevice, USER_READ, slot, send_data, 2, 1000);
```

uses the handle “hsbdevice” obtained from the open function, to send a user Read message as below

Start Condition	Target Address formed from the Board number of the device being used to send, and slot	Message type = USER_READ	Reply address formed from the data held in the handle “hsbdevice”	2 data bytes taken from send_data[0] and send_data[1]	Stop condition
-----------------	--	--------------------------	---	---	----------------

The parameter 1000 sets a timeout after which the function will return an error if the message has not been sent.

Calling the function receive message as follows :-

```
Status = HeHSBReceiveMessage(hsbDevice, &message_type, slot, reply, buffer_size, &bytes_read, 5000);
```

Will use the device “hsbdevice” (obtained from the open function) to receive a message. The function will not complete until a message has been received or the timeout value (5000ms) is reached. If a message is received from the “expected” master, i.e. that specified by forming an address out of the same board number as the host device being used, and “slot” parameters, the message type variable will be filled in, along with the values reply[0], reply[1] etc. The number of valid data bytes is shown by the variable “bytes_read”, which will not be allowed to exceed the value buffer_size, even if more data is actually returned in the HSB message.

For more details on these functions please refer to the Host-API user manual.

The more involved method breaks the message down into start, data and stop phases. Typically this is used for messages that have long data sections where it is not sensible to copy all of the data into a local array.

Again for details of these functions refer to the Host-API user manual

Appendix 1 – message definitions by Module class

C6000 Processor modules

Module Query and response

The processor modules need to respond to a request for their type and memory configuration etc.

Their response is :-

“Module type” is processor = 1.

HERON module number (1/2/3/4)

Processor option (1=C6201, 2=C6701, 3=C6203)

Firmware revision

Module option number -- normally 0, but allows for memory options etc

The exchange would then be :-

Master to processor module

start → module address → module type query (01) → address of requestor → stop

processor module to "original master"

start → address of requestor → module query response(02) → module address (from) → module type (01) → family number → Firmware revision → Processor option → option number → stop

Server/Loader run query

With HEPC9 and other HEART based boards, the Server/Loader will first boot the C6x processors via the FIFO's. The C6x's will then start to execute the booted program. The bootloader() function (mandatory in C6x programs to be loaded by the Server/Loader) will wait until a message arrives via HSB. This message tells the bootloader() function it should now complete (return).

The bootloader() functions running on the C6x processors do not respond to the run query, nor do they acknowledge the message in any way.

The exchange would then be :-

Master to processor module

start → module address → SL run type query (175) → FIFO number to read from server on host PC → FIFO number to write to server on host PC → stop

With the Server/Loader the 'Master' would be the host PC (ie the host bus's HSB node: 5).

FPGA modules/HERON_IO modules that have an FPGA

Module Query and response

The FPGA modules need to respond to the same request for type information.

The FPGA class of module can in fact also have I/O but any module of type FPGA will accept a user supplied bitstream hence no definition of the I/O related functions can be made by the module.

The information that they return is:-

"Module type" is FPGA = 2 (HERON-FPGA) or IO with FPGA = 4 (HERON-IO)

FPGA family number (1/2/3 etc)

Option/version code.

FPGA string (see below)

The exchange would then be :-

Master to FPGA module

start → module address → module type query (01) → address of requestor → stop

FPGA module to "original master"

start → address of requestor → module query response(02) → module address (from) → module type (02 or 04) → family number → option → String byte 0 → String byte1...String byte26 → stop

The string is the string that Xilinx put into their bitstream files. It is always 27 bytes long, but can actually be null terminated before that. E.g a virtex XCV50 would return v50bg256-6

The Option/version code allows HUNT ENGINEERING to differentiate between hardware options or versions that are the same board, but require different bitstreams. Usually this is set to 1, so a directory name is formed fpgaXvY or ioXvY where X is the family number and Y is the option/version number.

Configure FPGA

HERON-FPGA and HERON-IO modules must also receive requests to configure/reconfigure their FPGA, but this request can be made inherent in the bitstream transmission. i.e. a message that is of type "configure" can automatically generate the program pulse at the beginning.

In this way re-configuration from an on board prom can be requested by a message containing an empty bitstream message.

The Configuration transaction will be:-

Master to FPGA module

start→module address→Configure (03) →address of requestor→first config byte→ 2nd config byte.....last config byte→stop

FPGA module to original master

start→address of requestor→configuration success (05)/configuration fail(06) →module address→stop

Any further use of the HSB will be defined by the bitstream supplied to the module.

Configure FPGA with compressed format

HERON-FPGA modules with a family member of 4 or later and HERON-IO modules with a family member of 5 or later will respond to the special options request message. The first byte of the reply is 0 if a compressed configuration cannot be accepted and a 1 if it can. Actually the module types prior to those listed above can indicate that they do support compressed configuration in the last byte of their response from a module query.

If a module does support a compressed configuration sequence the Configuration transaction will be:-

Master to FPGA module

start→module address→Configure (13) →address of requestor→first config byte→ 2nd config byte.....last config byte→stop

FPGA module to original master

start→address of requestor→configuration success (05)/configuration fail(06) →module address→stop

Any further use of the HSB will be defined by the bitstream supplied to the module.

In fact the configuration sequence is not now each and every byte of the configuration, but byte pairs of :-

Number of bytes (1-255)

Value of byte (0-255)

Certainly any configuration that has areas of unused gates (like all of the HUNT IP and examples) will have large areas of zeros and hence this configuration sequence will complete much more quickly. For a full design it may actually take longer.

User Read and Write Access

The actual use of these messages cannot be defined here, but the format of them must be.

Master to FPGA module

start→module address→user write (08) →address of requestor →register address byte
→value →optional value byte→optional value byte..... →stop

In this way single or multiple bytes can be written, starting from the address given.

Nothing is returned from a write request.

This will result in the 8 bit address being written into the application FPGA using an address strobe, then one or more data bytes being written to the application FPGA using a data strobe, and qualified by a write signal. It is therefore the responsibility of the application FPGA to support auto incrementing addresses if required by its function.

For a read request

Master to FPGA module

start→module address→user read (09) →address of requestor →register address byte
→length byte→stop

In this way single or multiple bytes can be requested, starting from the address given.

The reply will be

FPGA module to original master

start→address of requestor→user read response(10) →module address→data
byte→optional data byte.... →stop.

This will result in the 8 bit address being written into the application FPGA using an address strobe, then one or more data bytes being read from the application FPGA using a data strobe, and qualified by the absence of write signal. It is therefore the responsibility of the application FPGA to support auto incrementing addresses if required by its function. The count of the data bytes read will be controlled by the controller (non application) FPGA.

Special Options and response

In order to support special features, this message has been added to the HERON-FPGA4 and FPGA modules designed after it. It has been made general purpose so that the same message type can be extended to support other features in the future. Modules before HERON-FPGA4, i.e. FPGA1, FPGA2, FPGA3, IO2 and IO4 do not and will never support this message type. Using this message type on one of those modules can cause the system to lock up requiring a system reset.

The information returned is:-

“Special Options Response” which is a series of 30 bytes. These bytes are defined as :-

First byte : Compressed bitstream support . 0=no, 1 = yes

Other bytes not used at this time.

The exchange would then be :-

Master to FPGA module

start → module address → Special Options request(11) → address of requestor → stop

FPGA module to "original master"

start → address of requestor → Special Options response(12) → module address (from) → options byte 0 → Options byte 1...Options byte 29 → stop

I/O only Modules

Module Query

I/O modules that support HSB may be configured using HSB. I/O modules are probably the module type that will have the least amount of real estate (logic) available for the HSB interface. This means that it is probably better to leave the actual definition of the options for each module type to the configuration software tool, rather than expect the module to provide a list.

They must reply with :-

"Module type" is IO only = 03

"Module number" 0/1/2/3.... reserve 255 to mean next byte is number (255+).

The exchange would then be :-

Master to IO module

start → module address → module type query (01) → address of requestor → stop

IO module to "original master"

start → address of requestor → module query response(02) → module address (from) → module type (03 → family number → optional second family number → stop

User Read and Write Access

Depending on the particular IO module there might be other configuration type messages that can be used.

The actual use of these messages cannot be defined here, but the format of them must be.

Master to IO module

start → module address → user write (08) → address of requestor → register address byte → value byte → optional value byte → optional value byte..... → stop

In this way single or multiple bytes can be written, starting from the address given.

Nothing is returned from a write request.

This will result in the 8 bit address being written to the IO module using an address, then one or more data bytes being written. It is therefore the responsibility of the IO module logic to support auto incrementing addresses if required by its function.

For a read request

Master to IO module

start→module address→user read (09) →address of requestor →register address byte
→length byte→stop

In this way single or multiple bytes can be requested, starting from the address given.

The reply will be

IO module to original master

start→address of requestor→user read response(10) →module address→data
byte→optional data byte.... →stop.

This will result in the 8 bit address being sent to the IO module. Then one or more data bytes will be read and returned to the master from the IO module. The count of the data bytes read will be controlled by the IO module implementation.

HEART FPGAs.

These would never need to respond to a request for information, nor to accept a bitstream for FPGA configuration.

There are not enough 7-bit addresses on a board to uniquely address each FPGA and each module. If we assign one address to the HEART FPGA, then the next whole byte can be the actual address.

The same technique can be used for configuring the routing table in the inter-board FPGA, which will have some extra configuration registers.

HEART Configuration

The protocol sent to the HEART FPGAs is simple, e.g. "register address"; "data"; "register address"; "data"; "register address"; "data" etc.

This allows a configuration program to change only those settings that have changed.

This also assumes 8 bit registers, but certain registers can be defined to expect "register address"; "data"; "data"; "data"; "data" e.g. a 32 bit register.

The exchange would then be :-

Master to HEART FPGA

start → HEART FPGA (7) → HEART FPGA number → HEART config (07) → register address → data → optional data1 → optional data2 → optional data3 → stop

For details of specific messages and register bit definitions refer to the user manual of a particular HEART carrier board.

HEART Expansion Modules

Module Query

The HEART "EM" modules need to respond to a request for their type and configuration etc.

Their response is :-

"Module type" is EM = 5.

EM module number (1/2/3/4)

Module option number -- normally 1, but allows for different functional programming of the same module, i.e. comport version of EM1.

The exchange would then be :-

Master to processor module

start → module address → module type query (01) → address of requestor → stop

processor module to "original master"

start → address of requestor → module query response(02) → module address (from)
→ module type (05) → family number → option number → stop

Make connection

The protocol sent to the HEART Expansion Modules is simple, e.g. "register address"; "data"; "register address"; "data"; "register address"; "data" etc.

This allows a configuration program to change only those settings that have changed.

This also assumes 8 bit registers, but certain registers can be defined to expect "register address"; "data"; "data"; "data"; "data" e.g. a 32 bit register.

The exchange would then be :-

Master to HEART Expansion Module

start → module address → HEART config (07) → register address → data → optional data1 → optional data2 → optional data3 → stop

For details of specific messages and register bit definitions refer to the user manual of a particular HEART Expansion Module.