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

Sending HSB Messages from the Contents of the Application FPGA on HERON-FPGA & HERON-IO Modules

Rev 2.0 R.Williams 24-10-03

The HERON-FPGA and HERON-IO families are ranges of HERON modules with FPGAs, often combined with some interface capability. The HERON-FPGA and HERON-IO families provide an FPGA that amongst many other things provides a connection to the HERON Serial Bus (HSB) communication interface.

Using HSB, messages can be sent from another system device such as a host interface or DSP module in order to set internal parameters of the Application FPGA function. All of the HERON-FPGA and HERON-IO modules can be used in this way.

In addition to being able to receive messages some of the HERON-FPGA and HERON-IO modules also have the capability to send messages over HSB. The content of these messages is defined by the user design placed in the application FPGA of these modules.

The message sending capability also enables embedded systems to directly configure the HEART communications system.

The HERON-FPGA1, HERON-FPGA2 and HERON-IO1 modules do not support the HSB 'mastering' function, but all modules designed after them can be expected to include this capability.

This document describes how to transmit HSB messages from the Application FPGA. Example VHDL source is supplied to demonstrate the HSB mastering functions. Please note, this VHDL source is generic and intended for use with all of the modules that support HSB mastering; this example does not include pre-compiled examples for each of the module types.

History

Rev 1.0 First written

Rev 2.0 Added sections on single message transfer

What is HSB?

The HERON Serial Bus (HSB) is a simple communication interface that forms part of the HERON Specification. It is intended to be used alongside HERON FIFO connections as a mechanism for transferring lower bandwidth set-up and control information.

That is, HERON systems are designed for real time operation using connections that are based upon FIFOs that guarantee no arbitration delay and the (maximum) bandwidth available. For system functions that do not require the use of the real-time FIFO connections, it would be a waste of these resources to use them. For these functions HSB can be used.

This tutorial shows how HSB can be used to send messages from the Application FPGA of your module to any other device connected to HSB. It is expected that in using this document you are already familiar with HSB and understand how HSB can be used to control parts of a system. If you need to refer to the HSB Specification you can do this using the HUNT ENGINEERING CD front-end and navigating to User Manuals -> Technology Documents.

HSB on Modules with FPGA

The HERON Serial Bus is not directly connected to the main FPGA of the module, because it is used to send the configuration information for the FPGA. For this reason the HSB is connected to the small control FPGA on the modules and a secondary interface is used to communicate between the two FPGAs on the module. You can refer to the specification of this secondary interface using the HUNT ENGINEERING CD front-end and navigating to User Manuals -> Technology Documents to access ‘Control FPGA - Application FPGA Message Interface specification’.

Normally though you would access the HSB from the main FPGA using the HE_USER interface of the Hardware Interface Layer.

Message 'Mastering' with HSB

HSB is a multi-mastered arbitrated bus that uses a protocol defined by HUNT ENGINEERING. It is built using a simple two-wire bus carried through two pins of the HERON module specification. As it is a multi-mastered bus, this means that any node on the bus can be used to send message data to any other node on the bus.

Typically for FPGA modules, HSB would be 'mastered' by the host interface in order to transfer the configuration bitstream to the application FPGA. However, as discussed in this document, HSB can be mastered by the contents of the application FPGA in order to transmit messages to other HSB nodes in the system. In addition, being able to master HSB from the application FPGA is particularly useful in an embedded system where there are no host interfaces or DSP modules that can be used to configure and control operation. In such systems it is necessary that the FPGA is used to control operation, an example of this being configuration of the HEART communication system on carriers such as the HEPC9.

In order to access HSB for message mastering, a user application must use the interface presented by the HE_USER component of the Hardware Interface Layer. Some of the signals included in the HE_USER interface are only used when the application FPGA is receiving messages and are not of interest here. The remaining signals that are used in HSB message mastering are shown below:

```
MSG_CLK          -- HSB clock
MSG_SEND_MSG     -- send command
MSG_DONE         -- used to indicate successful transmission
MSG_COUNT        -- message-sending counter enable
MSG_CLEAR        -- message-sending counter clear
MSG_CE           -- clock enable to control speed of operation
MSG_DOUT         -- data for transmission
MSG_SEND_ID      -- command to send module & slot ID information
MSG_LAST_BYTE    -- used to indicate when the last byte is presented
```

Sending a Message

To send a message over HSB, the signal MSG_SEND_MSG must be asserted (set high) for the whole duration of the message. When the MSG_SEND_MSG signal is first asserted, the HE_USER component will begin the process of mastering the bus so that message sending can be proceed. When the bus has become available for transmission, data transfer will begin.

The data to be transmitted must be presented on the data bus MSG_DOUT. As each byte of data is transmitted, the HE_USER component will indicate the next byte is required by asserting (driving high) the MSG_COUNT signal, for one cycle of the HSB clock signal MSG_CLK.

By using a memory component, a counter, and the MSG_COUNT signal, a very simple circuit can be created for data transmission. Firstly, the data values for transmission should be stored in a memory component such that the first memory location contains the first byte for transmission, and the last used memory location contains the last byte for transmission.

Alongside the memory should then be a counter that is reset to zero before transmission begins. The count enable of that counter should be directly driven by the MSG_COUNT signal, and the count value used to drive the address of the memory component.

When the MSG_SEND_MSG signal is first asserted, the counter will be at zero, pointing to the first location of memory containing the first transmission byte. As each byte is transferred, the MSG_COUNT signal will increment the address to automatically present the next data byte.

In addition to this process, extra logic is required to correctly drive the signals MSG_SEND_ID and MSG_LAST_BYTE. The HSB message format uses one data byte in each message to indicate the 'ID' of the HSB node that is sending the message. This ID information is based on the Module ID and Carrier ID information presented on the pins of the HERON module interface. The Module ID and Carrier ID information identify a particular HERON module slot on a particular carrier card in the user's system.

In order to transmit the appropriate ID information, the signal MSG_SEND_ID should be driven high in the data byte position in the message where the ID information is required. With the MSG_SEND_ID control set high, the HE_USER component will ignore the value presented on the MSG_DOUT data bus and replace it with the Module ID and Carrier ID information organised in the correct way.

As the last data byte of the message is being transmitted, the MSG_LAST_BYTE signal must be driven high to indicate to the HE_USER component that the end of the message has been reached. After the last byte has been transmitted, the MSG_SEND_MSG signal should be driven low to release HSB and complete the message sending process.

In order to correctly drive the MSG_SEND_ID and MSG_LAST_BYTE signals, the memory component used to store data bytes can also be used to store control bits that directly drive these signals. That is, by using a memory component that is at least 10 bits wide, two bits of control information can be stored at each location in parallel to the data for transmission. One of the extra memory locations can be set to define a data byte to be replaced by ID information and the other set high for the last byte of transmission.

At the end of message transmission, the MSG_DONE signal is driven high by the HE_USER component to indicate successful completion of the message.

At the end of message transmission, the MSG_CLEAR signal is also driven high and is intended for connecting to the asynchronous reset input of the counter logic described above. In this way, the address counter is automatically reset to the beginning of the message.

The MSG_CE signal acts as a clock enable for all HSB mastering logic inside the HE_USER component. The message mastering function inside the HE_USER component must operate at no more than 12.5MHz. With a typical HSB clock rate of 50MHz for example, this requires that the MSG_CE signal is driven high once every four clock cycles to achieve the operating frequency of 12.5MHz.

Sending Multiple Messages

The sending of multiple messages is achieved by repeating the single-message sending process over and over until all messages have been sent. When the first message is ready for transmission, the MSG_SEND_MSG signal should be asserted. At this point the HE_USER component will begin the process of obtaining control of HSB. When this process has completed data transfer will begin. The MSG_COUNT signal will be asserted as each byte is transferred. When the last data byte of the first message is reached the MSG_LAST_BYTE signal must then be asserted.

When the last data byte has been transferred the MSG_DONE signal will become asserted for one clock cycle. At this point the MSG_CLEAR signal will become asserted.

The user logic at this point must de-assert the MSG_SEND_MSG signal for a short period of time. There is one timing requirement that must be met between the sending of two messages. In order to ensure the correct operation of the HE_USER component, a 40us gap is required between one message and the next. This timing requirement is necessary to ensure that the Control FPGA connected to the Application FPGA is able to complete the message being sent and reach the correct bus state for the transmission of a new message.

This is simply achieved by de-asserting the MSG_SEND_MSG signal for 40us at the end of each message. The simplest way to implement the 40us period is to implement a counter that increments from zero following the assertion of MSG_DONE. When operating at 50MHz, this counter will need to reach the count of 2048 at which point the MSG_SEND_MSG signal can be re-asserted for the next message.

As with sending a single message an address counter and memory component can be used to create the source of message bytes for transmission. Where there are multiple messages to be transmitted, each message should be stored in the memory component such that the first message is at the beginning of memory, followed by the second message and so on up to the last message at the end of memory.

The first byte of each new message should stored be in the very next address location from the last byte of the previous message. In this way, the address counter incrementing on MSG_COUNT will automatically point to the first byte of the next new message. When messages are placed back to back in memory, the address counter should not use the MSG_CLEAR signal as this will reset the address to the beginning of memory and would be incorrect.

In addition to the eight data bits and two control bits described in the previous section, the memory component may also be used to store a third control bit. This third control bit can be used to indicate when the last byte of the last message has been reached in the memory. The control logic around the memory component can then detect when this bit goes high to stop the message sending process on completion of the last byte of the last message.

If you had multiple messages that need to be sent at different times during the operation of your system then you can store all of the messages in the same memory component and use top address bits to indicate which message or sequence of messages should be sent.

Of course you can design your own method for storing and sending multiple messages, but we have supplied VHDL sources that use the methods described above.

Assembling the Data Bytes for Transmission

Every message sent over HSB must use the data format defined in the HSB Specification. Accordingly, all data bytes transmitted through the HE_USER component must meet this specification.

Each message must start with a three-byte sequence that defines the address of the target HSB node, the message type, and the address of the sending HSB node. When the three control bytes have been sent, multiple optional data bytes may follow depending on the type of message being transmitted.

In order to correctly assemble a sequence of data bytes for HSB message transmission, the user will either use HUNT ENGINEERING tools to assemble the message or will need to assemble the message manually using the information contained in the HSB specification.

For users who wish to configure the HEART communication system from their FPGA design, a software tool is provided that will generate data in the correct format. This tool is the same program used to directly configure HEART from the host PC. The '`heartconf`' HEART configuration tool takes as input a network file defining the connections that must be made. With a correctly defined network file, `heartconf` should be run, specifying the '`-b`' option to generate message data for FPGA HSB-mastering. An output file will be created in the form of VHDL. This file can be directly inserted into an FPGA design, and combined with the example VHDL detailed in the following sections of this document.

Using the Example VHDL

The example VHDL comprises of three VHDL source modules that demonstrate HSB message mastering when used for HEART configuration. As such, the example VHDL is based around a multiple-message sending process.

The example VHDL provided is based around the use of a single Block RAM component. This Block RAM component must be initialised in a pre-defined way. In doing so, when the design is built the bit-stream will contain all of the HSB message data in the initialised Block RAM.

All that is required in addition to the Block RAM is logic that will control when to send each message that the memory component contains. It is up to the individual user what trigger mechanism is used to begin sending each message, although this document details what will need to be considered.

For those users who need to send a single HSB message the example VHDL can also be used as a starting point for development. In this case, the Block RAM component used to contain the message information will only need to contain one message rather than many.

The example VHDL modules interface directly with the HE_USER component in the Hardware Interface Layer in order to master HSB for sending messages.

The HSB_MSTR Component

The first of the three VHDL source modules is the source file '`hsb_mstr.vhd`'. This source file contains the entity `HSB_MSTR`. This entity forms the interface between your own application code and the HSB message mastering functionality.

In order to use the HSB mastering example code, this entity must be added into your design.

```
entity HSB_MSTR is
   port (
      RST          : in  std_logic;
      CLK          : in  std_logic;
      SEND_ONE_MSG : in  std_logic;
      RESTART      : in  std_logic;
      MSG_SENT     : out std_logic;
      END_OF_MSGS  : out std_logic;
      --
      DONE         : in  std_logic;
      COUNT        : in  std_logic;
      CLEAR        : in  std_logic;
      SEND_MSG     : out std_logic;
      CE           : out std_logic;
      DOUT         : out std_logic_vector(7 downto 0);
      SEND_ID      : out std_logic;
      LAST_BYTE    : out std_logic
   );
end HSB_MSTR;
```

The entity has two groups of input and output signals. This first group contains the signals `RST`, `CLK`, `SEND_ONE_MSG`, `RESTART`, `MSG_SENT` and `END_OF_MSGS`. The signals in this group are used to control when each message is transmitted and must be connected to signals generated by your own logic.

The second group contains the remaining signals `DONE`, `COUNT`, `CLEAR`, `SEND_MSG`, `CE`, `DOUT`, `SEND_ID`, and `LAST_BYTE`. These signals must be connected directly to the '`HE_USER`' HSB message interface signals of the Hardware Interface Layer.

The `HSB_MSTR` component is used to send one message at a time from a single Block RAM component, until there are no more messages left to send in the memory component.

Following a component reset, the message pointer is set to zero to point to the first message stored in the Block RAM. The component reset is the active high input signal `RST`. When asserted (set to 1) all of the HSB mastering logic will be reset.

Once the component has been initially reset, messages can be sent one at a time by pulsing the `SEND_ONE_MSG` signal. This signal must be set high for between one to four clock (`CLK`) cycles to cause the internal logic to start sending the next queued message. Once transmission of each message has completed the signal `MSG_SENT` will be asserted for four clock cycles. When the last message in the Block RAM has been sent, the `END_OF_MSGS` signal will be set high.

If the message transmission process needs to be restarted from the beginning of the Block RAM, then the RESTART signal should be asserted for one clock cycle to restart the internal logic.

Using the HSB_MSTR Component in Your Project

To use the HSB_MSTR component it must be added into your design project and instantiated in your design source. First copy the source file 'hsb_mstr.vhd' into the 'src' source directory of your design. Then use 'Project->Add Source' to add this file to the project.

Having done this declare the component below the architecture statement, but before the begin keyword as follows:

```
component HSB_MSTR
   port (
      RST          : in  std_logic;
      CLK          : in  std_logic;
      SEND_ONE_MSG : in  std_logic;
      RESTART      : in  std_logic;
      MSG_SENT     : out std_logic;
      END_OF_MSGS  : out std_logic;
      DONE         : in  std_logic;
      COUNT        : in  std_logic;
      CLEAR        : in  std_logic;
      SEND_MSG     : out std_logic;
      CE           : out std_logic;
      DOUT         : out std_logic_vector(7 downto 0);
      SEND_ID      : out std_logic;
      LAST_BYTE    : out std_logic );
end component;
```

Next instantiate the component in your design (below the begin statement) as follows:

```
iHSB_MSTR : HSB_MSTR
   port map (
      RST          => RESET,
      CLK          => CLOCK,
      SEND_ONE_MSG => SEND_ONE_MSG,
      RESTART      => RESTART,
      MSG_SENT     => MSG_SENT,
      END_OF_MSGS  => END_OF_MSGS,
      DONE         => MSG_DONE,
      COUNT        => MSG_COUNT,
      CLEAR        => MSG_CLEAR,
      SEND_MSG     => MSG_SEND_MSG,
      CE           => MSG_CE,
      DOUT         => MSG_DOUT,
      SEND_ID      => MSG_SEND_ID,
      LAST_BYTE    => MSG_LAST_BYTE );
```

At this point logic must be created to drive the input signals RESET, CLOCK, SEND_ONE_MSG, and RESTART. This logic should also make use of the output signals MSG_SENT and END_OF_MSGS. Please note, the remaining signals must be connected to the appropriate signals in the USER_AP interface.

An Example of Mastering HSB

Although it is entirely up to the user to decide how to trigger the sending of each individual message, the following example is provided to show how this might be done. In this example the system reset is used as the initial trigger to begin message sending. That is, when RESET is de-asserted message sending will begin after a short delay to allow any just reset logic to reach the correct state. At the completion of each message, the next is automatically started.

```
-- Catch the falling edge of RESET (reset becoming de-asserted)
process(CLOCK)
begin
   if rising_edge(CLOCK) then
      RES_DL_A <= RESET;
      RES_DL_B <= RES_DL_A;
      RES_DL_C <= '0';
      if RES_DL_A='0' and RES_DL_B='1' then
        RES_DL_C <= '1';
      end if;
   end if;
end process;
-- 4-bit up counter
process(RESET, CLOCK)
begin
   if RESET='1' then
      STARTcount <= "0000";
   elsif rising_edge(CLOCK) then
      if COUNTen='1' then
         STARTcount <= STARTcount + 1;
      end if;
   end if;
end process;
-- Count for 16 cycles
TC <= '1' when STARTcount="1111" else '0';
process(RESET, CLOCK)
begin
   if RESET='1' then
      COUNTen <= '0';
   elsif rising_edge(CLOCK) then
      COUNTen <= '0';
      if (RES_DL_C='1') or (COUNTen='1' and TC='0') then
         COUNTen <= '1';
      end if;
   end if;
end process;
-- Start sending messages when counter=15, and keep going until end of messages
SEND_ONE_MSG <= TC OR (MSG_SENT AND (NOT END_OF_MSGS));
-- Tie unused MSG_READY signal low
MSG_READY <= '0';
```

The Block RAM Component

There are two other VHDL source modules that are provided in addition to '`hsb_mstr.vhd`'. Only one of these files will be required by your project depending on the FPGA type you are using.

If the project is being built for a Spartan-II FPGA then you will need to include the source file '`smsg_ram.vhd`'. Alternatively if you are using a Virtex-II FPGA you will need to include the source file '`vmsg_ram.vhd`'. To include the appropriate file, first copy the file to the '`src`' directory of your project and then add it using '`Project->Add Source`'.

These files contain a single entity '`MSG_RAM`'. This entity is used inside the `HSB_MSTR` component as the source of the Block RAM component that must contain the sequence of messages.

The Block RAM files that are provided as part of this example simply form 'templates' for the Block RAM instantiation and initialisation of its contents. The exact application specific content of these files can be generated in one of two ways.

The first method is to use an automatically generated Block RAM file, generated by the HUNT ENGINEERING utility '`heartconf`'. This method is appropriate where the HSB mastering function is to be used to configure the HEART connections of a HERON carrier such as the HEPC9.

The second method is to hand-edit the appropriate file in order to create the desired sequence of bytes for transmission. When doing this you will need to understand the format of HSB message bytes along with the format used to store those bytes in the Block RAM. This method is appropriate when you are not using the Block RAM for configuring HEART but to send more general messages from one node to another.

Method 1: Auto-generation using HEARTCONF

If you are using the HSB mastering example to configure HEART on a suitable HERON module carrier then you must generate the `vmsg_ram.vhd` or `smsg_ram.vhd` file automatically using `HEARTCONF`.

To do this first edit the network file you are using to correctly specify the connections to be made by HEART. When you have done this, run `HEARTCONF` with the correct Block RAM auto-generate switches set. Please refer to the User Manual for the `HEARTCONF` Tool to ensure that you have correctly specified the auto-generate switches, ensuring that the file you generate is for the correct FPGA type.

When this file has been generated, copy it into the '`src`' directory of your project, making sure that it is added to the project source only once.

Method 2: Hand Assembling the Block RAM Initial Values

If you are not using the HSB mastering example to configure HEART, then it will be necessary to hand assemble the Block RAM initialisation values in the component, `MSG_RAM`.

If you are going to do this you will need to understand the process of initialising Block RAM components in VHDL. If you are unsure how to do this, please refer to the appropriate Xilinx documentation on this subject.

The Block RAM needs to be initialised with a sequence of 'words' that contain successive bytes for transmission over HSB along with control information. The first byte to be transmitted must be stored at address zero in the memory, with consecutive bytes being stored at subsequent addresses in the memory.

The memory is organised as 16-bit wide memory. For each 16-bit value, the bottom byte (the 'data-byte') is the value that will be transmitted on HSB and the top byte (the 'control-byte') is used to carry control information.

Therefore the 16-bit value stored at address 0 will contain the first data byte in the bottom half of the word, and the first control byte in the top half of the word. The 16-bit value stored at address 1 will contain the second data byte and the second control byte.

Assembling the Control Byte

The following table defines the function of each bit contained in the Control Byte.

Control-Byte Bit	Bit Function
0	Set to 1 to indicate that the modules' own address must be transmitted in place of the associated data byte. Only set this bit high at an address where the module address is required.
1	Set to 1 to indicate the last byte in the current message. This bit must only be set high at an address containing the last byte of a message.
2	Set to 1 to indicate the last message in total. This bit must be set high at the address of the last byte in the last message, AND at the very next address location.
3 to 7	These bits are undefined. Always set to zero.

Example Message Sequence

In the following example we want to send two messages. The first message must be sent to a module in Slot 1 of Board 0, and the second message must be sent to a module in Slot 3 of Board 0.

The first message (for Slot 1) is a message of message-type '8', with the data bytes to follow being 01h 02h 03h and 04h.

The second message (for Slot 3) is a message of message-type '12', with the data bytes to follow being 11h 12h 13h and 14h.

According to the HSB Specification, a message starts with an 'address' byte, followed by a message type byte and then a data byte contained the address of the sender. After these a number of optional data bytes may be sent. Please note, for the Data-Byte value that is to form the first 'address' byte on the HSB bus, the value must be left shifted by 1.

The 16-bit hex memory initialisation values for this example will therefore be (starting at Block RAM address 0, and incrementing after that):

```
0002h, 0008h, 0100h, 0001h, 0002h, 0003h, 0204h,
0006h, 000Ch, 0100h, 0011h, 0012h, 0013h, 0614h, 0400h
```

All values after these must be set to zero.

Block RAM Sizes for the Spartan-II and Virtex-II FPGAs

The HSB mastering example uses a single Block RAM component to store the entire sequence of messages for transmission. When using the HSB mastering example it is important to understand the total available space of message data storage that is provided by the Block RAM component.

For the Spartan-II FPGA a single Block RAM component can store 4096 bits. Due to the differences between Xilinx families, it is possible to store four times this amount (16384 bits) in a Virtex-II Block RAM component.

Due to the fact that only a limited number of bytes can be stored in one Block RAM component, you will need to ensure that there is enough space to store all of the bytes required by your application. Where one Block RAM component is insufficient you will need to modify the code to increase the storage that is available.

Using the example VHDL it is possible to store 256 message bytes when using a single Spartan-II Block RAM, and 1024 message bytes when using a single Virtex-II Block RAM.