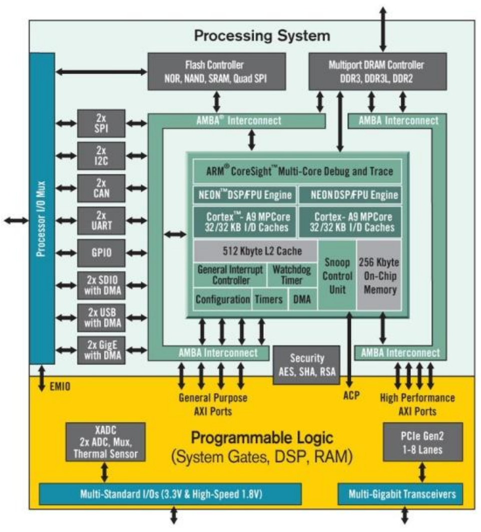
Designing a Custom AXI-lite Slave Peripheral

Version 1.0, July 2014
Rich Griffin (Xilinx Embedded Specialist, Silica EMEA)

Overview


Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Xilinx Zynq-7000 All Programmable SoC devices offer architectural features which offer designers an advanced ARM based processing system, coupled with a flexible region of Programmable Logic. The Xilinx Vivado tool suite offers the designer a powerful and flexible design environment, including support for designing and testing custom IP blocks which can be used, and re-used, in multiple designs.

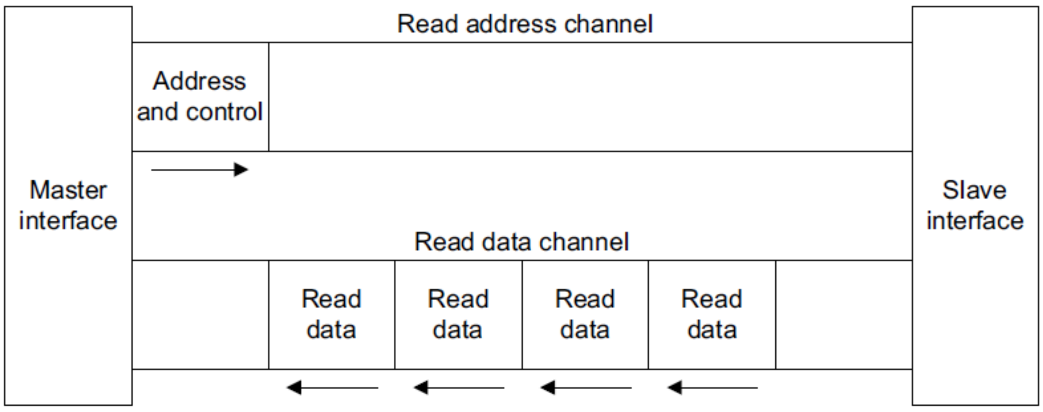
The Processing System (PS) includes two Cortex-A9 processor cores, a dedicated DDR memory controller, and a wide selection of IO peripherals. The Programmable Logic (PL) is based on the Xilinx 7-Series FPGA fabric and offers the designer the ability to implement their own custom logic which can work alongside the software running on the processor cores. The two areas of the device, PS and PL, are linked by a series of interfaces which adhere to the AXI4 interconnect standard. These interfaces allow the designer to implement custom logic in the PL which can be connected to the PS and extend the range of peripherals which are available and visible in the processor’s memory map. One of the most common uses of this technology is to create a block of custom logic in the PL, and then add control and status monitoring capabilities by using memory mapped registers which the processors can access via the AXI4 interconnect. The high performance of dedicated custom logic can then be utilised in the system, without sacrificing the flexibility of software for control and status monitoring tasks.

The AXI4 (Advanced eXtensible Interface) protocol has been designed to offer different variants of the interconnect. The full AXI4 specification offers a diverse range of powerful features including variable data and address bus widths, high bandwidth burst operations, advanced caching support, out of order transaction completion, and various transaction protection and access permissions. Although this specification provides the user with enormous flexibility and control, it is often desirable to implement a much simpler peripheral which uses only a subset of these features. For that reason, a reduced feature variant of the AXI4 specification exists in the form of “AXI4-lite”. This subset of the specification supports uses cases where only basic interconnect transactions are required, and removes some of the more advanced capabilities of the interconnect such as cache support, burst support, and variable bit widths for the address and data buses. The AX4-lite interconnect is therefore perfect for applications where simple control and status monitoring capabilities are required for a custom built IP block. This guide discusses how to build a custom IP block in the PL, implement memory mapped registers, and make them available via the AXI4-lite interconnect so that they are visible to the processor. This guide will also discuss the creation of some basic device drivers, showing how software can be written to access the registers on the custom peripheral.

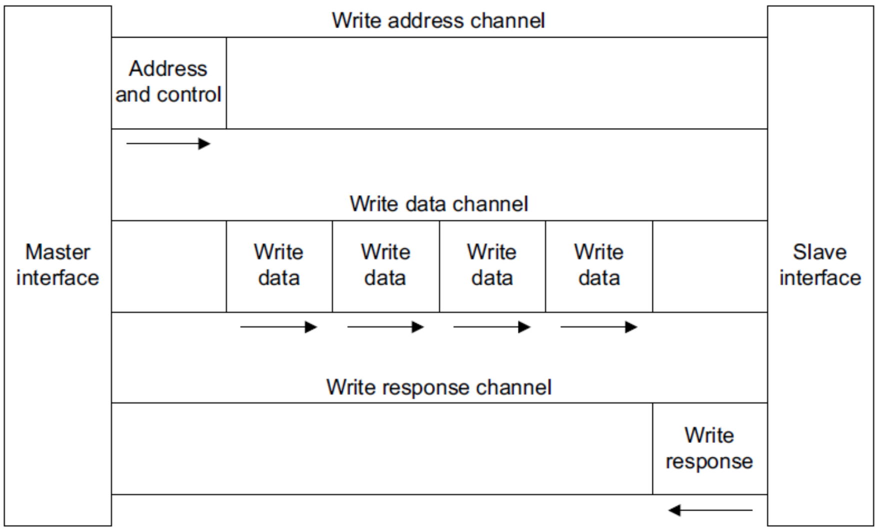
AXI4-lite Protocol; The Five Channels

The AXI4-lite protocol has been written with a modular design flow in mind. There are five channels which make up the specification; the read address channel, the write address channel, the read data channel, the write data channel, and the write acknowledge channel. It is important to understand the function of each of these channels before starting to develop the custom IP block. For the purposes of clarity, this guide discusses the concepts of read and write transactions from the processor's perspective.

The read address channel allows the processor to signal that it wishes to initiate a read transaction. As the name suggests, this channel carries addressing information and some handshaking signals. The read data channel carries the data values that are transferred during a transaction, along with their associated handshaking signals. Together, these two channels contain everything that is needed for a successful read transaction.



The write address channel allows the processor to signal that it wishes to initiate a write transaction. It is identical to the read address channel in every other way. The write data channel performs an equivalent function to be read data channel, except that the data flows in the opposite direction (i.e. towards the slave). For write transactions an additional channel is used in the form of the Write Response Channel. This last channel is used to allow the slave peripheral to acknowledge receipt of the data, or to signal that an error has occurred.



The next important aspect of AXI4-lite that must be understood is the handshaking signals. These signals are consistent across the five channels, and offer the user a simple yet powerful way to control all read and write transactions. The signals are based on a simple "Ready" and "Valid" principle; "Ready" is used by the recipient to indicate that it is ready to accept a transfer of a data or address value, and "Valid" is used to clarify that the data (or address) provided on that channel by the sender is valid so that the recipient can then sample it.

Taking the example of the Write Address Channel, the following signals are used:

AXI4-lite Read Address Channel			
Signal Name	Size	Driven by	Description
S_AXI_ARADDR	32 bits	Master	Address bus from AXI interconnect to slave peripheral.
S_AXI_ARVALID	1 bit	Master	Valid signal, asserting that the S_AXI_AWADDR can be sampled by the slave peripheral.
S_AXI_ARREADY	1 bit	Slave	Ready signal, indicating that the slave is ready to accept the value on S_AXI_AWADDR.

It is important to understand that a "Valid" signal can be asserted before the recipient is ready to receive it, but when this scheme is used the value presented by the sender must be held in the "Valid" state until the receiver is ready to proceed with the transaction. It is equally acceptable for the receiver to indicate a "Ready" state before the sender makes a value "Valid". In this latter case, the sender can be assured that the transfer will complete within one clock cycle, because it has forewarning of the receiver's readiness.

A frequently misunderstood use of the Valid and Ready signals, and one which often results in incorrect and illegal implementations of the AXI4-lite protocol, is the assumption that the sender can/must wait for "Ready" to be asserted by the receiver before it asserts its "Valid" signal. This is an illegal use of the handshaking signals and can result in a deadlock situation arising. Ready can be asserted before Valid, but the sender must never wait for Ready as a pre-condition to commencing the transaction.

This important aspect of the AXI4-lite protocol can be easily remembered by applying the "Assert and wait" rule. Never use the "Wait before Assert" approach, because this is illegal.

"Assert Ready and wait for Valid" ✓

"Assert Valid and wait for Ready" ✓

"Wait for Ready before asserting Valid" ✗

This is an important learning point. Make sure that this concept of handshaking is clearly understood before continuing.

The use of the Ready / Valid handshaking scheme is identical across all five AXI4-lite channels.

Read Transactions

Starting with the example of a read transaction, the signals belonging to the Read Data channel can now be examined.

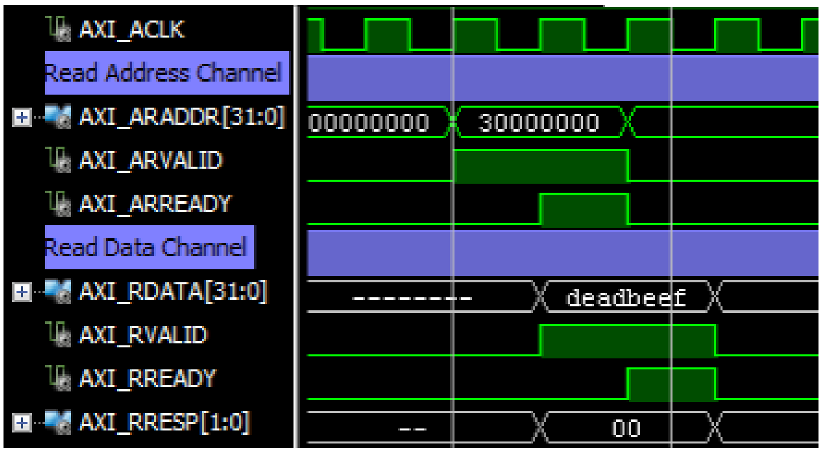
AXI4-lite Read Data Channel			
Signal Name	Size	Driven by	Description
S_AXI_RDATA	32 bits	Slave	Data bus from the slave peripheral to the AXI interconnect.
S_AXI_RVALID	1 bit	Slave	Valid signal, asserting that the S_AXI_RDATA can be sampled by the Master.
S_AXI_RREADY	1 bit	Master	Ready signal, indicating that the Master is ready to accept the value on the other signals.
S_AXI_RRESP	2 bits	Slave	A "Response" status signal showing whether the transaction completed successfully or whether there was an error.

The transaction in this example is one of the Master (e.g. the processor) reading a data value from the Slave (e.g. the peripheral). The important difference to recognise here is that the roles of driving Valid and Ready have now been reversed. For the purposes of data flow, the sender is now the Slave and the receiver is now the Master. Therefore, the Slave should provide the data value on S_AXI_RDATA and assert S_AXI_RVALID to qualify it. A new signal is introduced at this stage, S_AXI_RRESP, which is a status signal that describes whether or not the transaction has completed successfully. The Master can then assert S_AXI_RREADY to indicate that it is ready to accept the data (or, optionally, the Master could assert S_AXI_RREADY prior to the assertion of S_AXI_RVALID, indicating that it is ready to receive the data as soon as S_AXI_RVALID is asserted by the slave).

The coding of the S_AXI_RRESP signal should be discussed at this point, so as to understand its purpose. In most situations the RRESP signal will be tied to "00" because the more advanced features will not be needed. However, some of the more advanced functionality of the RRESP signals may occasionally be useful, usually when then design of the peripheral needs to signal a "retry" condition due to a delay in the availability of the data.

AXI4-lite Response Signalling		
RRESP State [1:0]	Condition	Description
00	OKAY	**"OKAY"** The data was received successfully, and there were no errors.
01	EXOKAY	**"Exclusive Access OK"** This state is only used in the full implementation of AXI4, and therefore cannot occur when using AXI4-Lite.
10	SLVERR	**"Slave Error"** The slave has received the address phase of the transaction correctly, but needs to signal an error condition to the master. This often results in a retry condition occurring.
11	DECERR	**"Decode Error"** This condition is not normally asserted by a peripheral, but can be asserted by the AXI interconnect logic which sits between the slave and the master. This condition is usually used to indicate that the address provided doesn't exist in the address space of the AXI interconnect.

The timing diagram for a complete read transaction, showing the use of the Read Address and Read Data channels, is shown below. In this example the Valid signals are asserted before the Ready signals on both channels, and it can be clearly seen that the transaction on each channel completes one clock cycle after Ready is asserted. If the Ready signals were pre-emptively asserted by the receiver in each case, the entire transaction could feasibly be shortened by two clock cycles, leaving the Valid signals asserted for just one clock cycle each. The use of the "00" AXI_RRESP state can also be clearly seen, indicating that the transaction completed successfully.



Write Transactions

Write transactions are almost identical to the Read transactions discussed above, except that the Write Data Channel has one signal that is different to the Read Data Channel. The S_AXI_WSTRB (Write STRoBe) controls which of the bytes in the data bus are valid and should be read by the Slave. For example, if the Master is performing an 8 bit write transaction to the slave then not all 32 bits of the data bus will be relevant. In the case of an AXI4-lite transaction the data bus is 32 bits wide, and therefore the WSTRB signal is four bits wide with each bit representing one byte (8 bits) of data.

AXI4-lite Write Data Channel			
Signal Name	Size	Driven by	Description
S_AXI_WDATA	32 bits	Master	Data bus from the Master / AXI interconnect to the Slave peripheral.
S_AXI_WVALID	1 bit	Master	Valid signal, asserting that the S_AXI_RDATA can be sampled by the Master.
S_AXI_WREADY	1 bit	Slave	Ready signal, indicating that the Master is ready to accept the value on the other signals.
S_AXI_WSTRB	4 bits	Master	A "Strobe" status signal showing which bytes of the data bus are valid and should be read by the Slave.

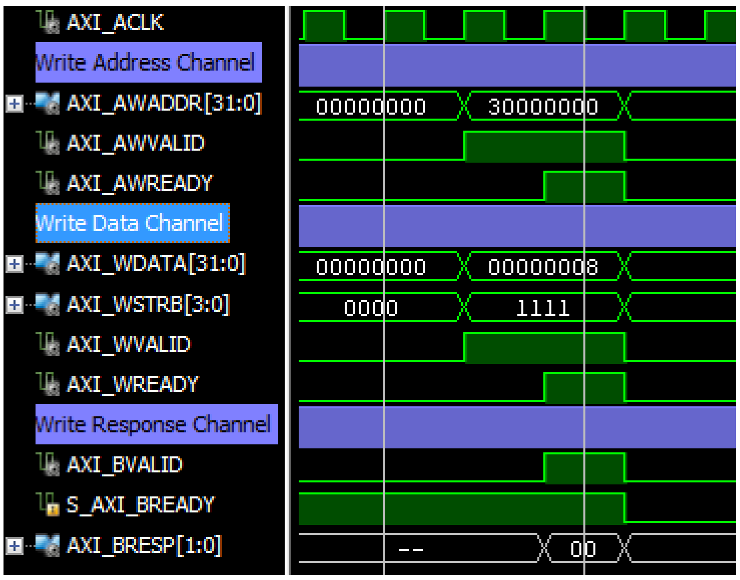
The write strobe signals can often be a source of confusion, specifically around which strobe signals relate to which bits of the data bus. The AXI specification describes this using an accurate and unequivocal formula "WSTRB[n] corresponds to WDATA[(8 × n) + 7:(8 × n)]", but sadly this is not always easy to comprehend at first glance. The following table serves to illustrate some practical examples of strobe usage. For the purposes of these (non exhaustive) examples, the "active" bits are shown as logic 1, and the "inactive" bits are shown as logic 0.

S_AXI_WSTRB signals		
S_AXI_WSTRB [3:0]	S_AXI_WDATA active bits [31:0]	Description
1111	11111111111111111111111111111111	All bits active
0011	00000000000000001111111111111111	Least significant 16 bits active
0001	00000000000000000000000011111111	Least significant byte (8 bits) active.
1100	11111111111111110000000000000000	Most significant 16 bits active

During a write transaction, the last of the five AXI channels (The Write Response Channel) is also used because the Slave needs a way to signal to the Master that the data was received correctly. The same Ready / Valid handshaking style is used as in the other channels. A simple 2 bit response system is used to signal a successful transaction or an error, coded in the same way as with the other RESP signals (e.g. "00" for a successful transaction, and similar codings for the error status conditions).

AXI4-lite Write Response Channel			
Signal Name	Size	Driven by	Description
S_AXI_BREADY	1 bit	Master	Ready signal, indicating that the Master is ready to accept the "BRESP" response signal from the slave.
S_AXI_BRESP	2 bits	Slave	A "Response" status signal showing whether the transaction completed successfully or whether there was an error.
S_AXI_BVALID	1 bit	Slave	Valid signal, asserting that the S_AXI_BRESP can be sampled by the Master.

A timing diagram of a complete AXI4-lite write transaction is shown below, showing the use of the three AXI write channels. In this example the value of 0x00000008 is being written to address 0x30000000. Note that this is a 32 bit write, and all four write strobe (WSTRB) signals are asserted. The use of the handshaking signals on all three channels is shown below, using the same "valid before ready" scheme on the address and data channels, and a "ready before valid" scheme on the write response channel. The two different handshaking schemes are not used here for any specific reason, other than to demonstrate an example of each style. Both are perfectly acceptable, and can be used in combination across the five AXI4-lite channels.



AXI4-lite Signal Names

The AXI4-lite signal names are completely flexible from the point of view of the VHDL design. During the creation of a Xilinx IP block, the Vivado tools can be used to map each AXI signal onto the signal name that the designer used when creating the IP. However in order to make the life of the designer much easier, the signal names shown here are recommended when designing a custom AXI slave in VHDL. Using these signal names will allow the Vivado design tools to automatically detect the signal names during the "create and package IP" step (described later on). This will save time, effort, and confusion when debugging the design.

```
-- Clock and Reset
S_AXI_ACLK        : in std_logic;
S_AXI_ARESETN     : in std_logic;
-- Write Address Channel
S_AXI_AWADDR      : in  std_logic_vector(31 downto 0);
S_AXI_AWVALID     : in  std_logic;
S_AXI_AWREADY     : out std_logic;
-- Write Data Channel
S_AXI_WDATA       : in  std_logic_vector(31 downto 0);
S_AXI_WSTRB       : in  std_logic_vector(3 downto 0);
S_AXI_WVALID      : in  std_logic;
S_AXI_WREADY      : out std_logic;
-- Read Address Channel
S_AXI_ARADDR      : in  std_logic_vector(31 downto 0);
S_AXI_ARVALID     : in  std_logic;
S_AXI_ARREADY     : out std_logic;
-- Read Data Channel
S_AXI_RDATA       : out std_logic_vector(31 downto 0);
S_AXI_RRESP       : out std_logic_vector(1 downto 0);
S_AXI_RVALID      : out std_logic;
S_AXI_RREADY      : in  std_logic;
-- Write Response Channel
S_AXI_BRESP       : out std_logic_vector(1 downto 0);
S_AXI_BVALID      : out std_logic;
S_AXI_BREADY      : in  std_logic;
```

Address Decoding

Another important concept to understand before creating an AXI4-lite custom slave peripheral is the address decoding. In previous versions of the Xilinx design flow (where PLB and OPB peripherals were typically used) it was necessary for each IP peripheral connected to the processor to individually decode all transactions that were presented by a master on the bus. This was due to the PLB and OPB buses being designed with so-called "multi-drop" architectures. In essence, all of the address, data, and handshaking signals in the previous PLB/OPB implementation were routed in parallel to each slave IP connected to the bus, and it was the responsibility of each peripheral to accept or reject each bus transaction depending on the address that was placed on the address bus. With AXI4-lite, the interconnect does not use a multi-drop architecture, but uses a scheme where each transaction from the master(s) is specifically routed to a single slave IP depending on the address provided by the master. This premise permits a completely different design methodology to be adopted by the creator of a slave IP, in that any transactions which reach the slave's interface ports are already known to be destined for that peripheral. As such, it is not necessary for functionality to be added to each slave IP to reject AXI4-lite transactions that are outside the addressable range of the slave IP. The designer merely needs to decode enough of the incoming address bus to determine which of the registers in the slave IP should be read or written.

An example of how this might be coded in VHDL is provided here for reference, showing how the incoming S_AXI_AWADDR bus might be decoded to assert a series of write enable signals for a large number of other registers in the design (98 registers in this example), without requiring large quantities of VHDL code to be written.

```
local_address <= to_integer(unsigned(S_AXI_AWADDR(31 downto 0)));

address_range_analysis : process (local_address)
begin
    manual_mode_control_register_address_valid <= '0';
    manual_mode_data_register_address_valid <= '0';
    servo_position_register_address_valid <= (others => '0');
    low_endstop_register_address_valid <= (others => '0');
    high_endstop_register_address_valid <= (others => '0');

    case (local_address) is
       when 0 => manual_mode_control_register_address_valid <= '1';
       when 4 => manual_mode_data_register_address_valid <= '1';
       when 128 to 252 =>
          servo_position_register_address_valid((local_address-128)/4) <= '1';
       when 256 to 380 =>
          low_endstop_register_address_valid((local_address-256)/4) <= '1';
       when 384 to 508 =>
          high_endstop_register_address_valid((local_address-384)/4) <= '1';
       when others => NULL;
    end case;
end process;
```

Implementing Addressable Registers

Using the address decoding scheme above, it is extremely simple to implement registers in VHDL which can receive data values written by a master on the AXI4-lite interconnect. The following extract of code shows how an individual register can be quickly and easily implemented (in this case mapped to BASEADDR + 0x00, as has been coded in the previous VHDL snippet).

```
manual_mode_control_register_process : process (S_AXI_ACLK)
begin
    if (S_AXI_ACLK'event and S_AXI_ACLK = '1') then
       if Local_Reset = '1' then
          manual_mode_control_register <= (others => '0');
       else
          if (manual_mode_control_register_address_valid = '1') then
              manual_mode_control_register <= S_AXI_WDATA;
          end if;
       end if;
    end if;
end process;
```

For read transactions, the data stored in the registers has to be routed back to the S_AXI_RDATA bus, and this can also be achieved very easily in VHDL. Here is an example showing how to implement read transactions from a series of registers, and the associated address decoding that is required to do so.

```
send_data_to_AXI_RDATA : process (     send_read_data_to_AXI, local_address,
                                servo_position_register_array,
                                manual_mode_control_register, manual_mode_data_register,
                                low_endstop_register_array, high_endstop_register_array)
begin
    S_AXI_RDATA <= (others => '-');
    if (local_address_valid = '1' and send_read_data_to_AXI = '1') then
        case (local_address) is
            when 0 =>
                S_AXI_RDATA <= manual_mode_control_register;
            when 4 =>
                S_AXI_RDATA <= manual_mode_data_register;
            when 128 to 252 =>
                S_AXI_RDATA <= X"000000" & servo_position_register_array((local_address-128)/4);
            when 256 to 380 =>
                S_AXI_RDATA <= low_endstop_register_array((local_address-256)/4);
            when 384 to 508 =>
                S_AXI_RDATA <= high_endstop_register_array((local_address-384)/4);
            when others => NULL;
        end case;
    end if;
end process;
```

Controlling the AXI Transactions

With all of the aforementioned building blocks in place, it simply remains to implement some logic to control the AXI transactions. This can be achieved by the use of a finite state machine. Here is an example of a (simplified) state machine, showing the implementation of some of the states, and showing how a read transaction might be handled in the design. The example is not designed to cover all of the states required to implement read and write transactions, but should help to illustrate a style of coding suitable for creating the FSM.

```
state_machine_update : process (S_AXI_ACLK)
    begin
        if S_AXI_ACLK'event and S_AXI_ACLK = '1' then
            if Local_Reset = '1' then
                current_state <= reset;
            else
                current_state <= next_state;
            end if;
        end if;
    end process;

state_machine_decisions : process (current_state, combined_S_AXI_AWVALID_S_AXI_ARVALID, S_AXI_ARVALID,
S_AXI_RREADY, S_AXI_AWVALID, S_AXI_WVALID, S_AXI_BREADY, local_address, ...{signals removed}... )
begin
case current_state is
        when reset =>
                next_state <= idle;
        when idle =>
                next_state <= idle;
                case combined_S_AXI_AWVALID_S_AXI_ARVALID is
                        when "01" => next_state <= read_transaction_in_progress;
                        when "10" => next_state <= write_transaction_in_progress;
                        when others => NULL;
                end case;

        when read_transaction_in_progress =>
                next_state <= read_transaction_in_progress;
                S_AXI_ARREADY <= S_AXI_ARVALID;
                S_AXI_RVALID <= '1';
                S_AXI_RRESP <= "00";
                if S_AXI_RREADY = '1' then
                        next_state <= complete;
                end if;

                case (local_address) is
                        when 0 => S_AXI_RDATA <= manual_mode_control_register;
                        when 4 => S_AXI_RDATA <= manual_mode_data_register;
                        when others =>
                            if (local_address >=8 and local_address < (8+((NUMBER_OF_SERVOS-1)*4))) then
                               S_AXI_RDATA <= servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4);
                            else
                                S_AXI_RDATA <= (others => '0');
                            end if;
                    end case;
        ... {additional code removed from here} ...
        when complete =>
                case combined_S_AXI_AWVALID_S_AXI_ARVALID is
                        when "00" => next_state <= idle;
                        when others => next_state <= complete;
                end case;


        when others => next_state <= reset;
end case;
end process;
```

Testing and Debugging

During the development of an AXI peripheral, it is essential to make sure that the AXI protocol is being implemented by the custom IP. If an incorrectly behaving IP is connected to an operational AXI interconnect within a Zynq device, the user may notice completely unpredictable behaviour from the interconnect. For example, if transactions are not correctly completed using the response signals, the interconnect may wait for long periods of time before timeout or retry conditions are met. It is almost impossible to trace the source of such problems inside the running silicon, because the user will have no visibility of the fault. A common mistake is for the user to quickly write some software to test the new peripheral, thinking that it will be a reliable testbench. Unfortunately the software might also have faults / bugs which need to be corrected, or there might be any one of a dozen system configuration settings which may have been forgotten. A situation can therefore easily arise where the user knows that a problem exists but doesn't know where to look to fix it, and simply makes the debugging task harder by introducing more complexity into the task.

A much better approach is to check for the correct operation of the peripheral before the additional complexity of software is bought into the equation. Keeping the test and debugging effort as simple as possible is a vital step when embarking upon the task of custom peripheral debugging. The easiest way to achieve this is to use a simple HDL simulation model which can be configured to generate AXI transactions and then observe the behaviour of the custom peripheral in the simulation waveform viewer. Common faults can be very quickly identified, and overall development time can be massively reduced.

The modular nature of the AXI protocol can be hugely beneficial when generating testbench models, because each of the five AXI channels can be implemented separately. An example of the VHDL code required to implement a model for the AXI read address channel is shown for reference. The code shown is not complete, but offers guidance of how the model could be implemented. Similar models can be written for the other AXI channels, and then be incorporated into a full testbench.

```
entity AXI_ADDRESS_CONTROL_CHANNEL_model is
   PORT
        (
        Clk         : in  STD_LOGIC;
        resetn       : in STD_LOGIC;
        go           : in STD_LOGIC;
        done         : out STD_LOGIC;
        address      : in std_logic_vector(31 downto 0);
        AxADDR     : out  STD_LOGIC_VECTOR (31 downto 0);
        AxVALID    : out  STD_LOGIC;
        AxREADY    : in   STD_LOGIC
        );
end AXI_ADDRESS_CONTROL_CHANNEL_model;

architecture Behavioral of
AXI_ADDRESS_CONTROL_CHANNEL_model is

type main_fsm_type is (reset, idle, running, complete);
signal current_state, next_state : main_fsm_type :=
reset;
signal address_enable : std_logic;

begin

AxADDR <= address when address_enable = '1' else (others
=> '0');

state_machine_decisions : process (current_state, go,
AxREADY)
begin
    done <= '0';
    address_enable <= '0';
    AxVALID <= '0';

    case current_state is
        when reset => next_state <= idle;

        when idle =>
        next_state <= idle;
        if go = '1' then
            next_state <= running;
        end if;

        when running =>
            next_state <= running;
            address_enable <= '1';
            AxVALID <= '1';
            if AxREADY = '1' then
```

Completing the Testbench

In this example, the five HDL models for the AXI channels have been instantiated inside a top level HDL wrapper file. An additional state machine to control all of the five channel models has been added, to enable them to work in unison. This method of implementation also allows the user to write very simple test stimulus in their test environment, and the AXI models will generate reliably correct interconnect transactions automatically. The following code shows an example of how the five AXI models can be controlled to work together. Each model has a "go" and a "done" signal, allowing them to be controlled and monitored. The user simply provides values on signals called "address", "go", and "RNW", and the data is transferred on "read_channel_data" and "write_channel_data" signals.

```
state_machine_decisions : process (current_state, read_transaction_finished,
write_transaction_finished, go, RNW, address, write_data, read_channel_data)

begin

write_channel_data <= write_data;
transaction_address <= address;
start_read_transaction <= '0';
start_write_transaction <= '0';
send_write_data <= '0';
busy <= '1';
done <= '0';

case current_state is
        when reset =>
                next_state <= idle;

        when idle =>
                next_state <= idle;
                busy <= '0';
                if go = '1' then
                        case RNW is
                                when '1' => next_state <= read_transaction;
                                when '0' => next_state <= write_transaction;
                                when others => NULL;
                        end case;
                end if;

        when read_transaction =>
                next_state <= read_transaction;
                start_read_transaction <= '1';
                if read_transaction_finished = '1' then
                        next_state <= complete;
                end if;

        when write_transaction =>
                next_state <= write_transaction;
                start_write_transaction <= '1';
                send_write_data <= '1';
                if write_transaction_finished = '1' and write_data_sent = '1' then
                        next_state <= complete;
                end if;

        when complete =>
                next_state <= complete;
                done <= '1';
                if go = '0' then
                        next_state <= idle;
                end if;

        when others =>
                next_state <= reset;
end case;
end process;
```

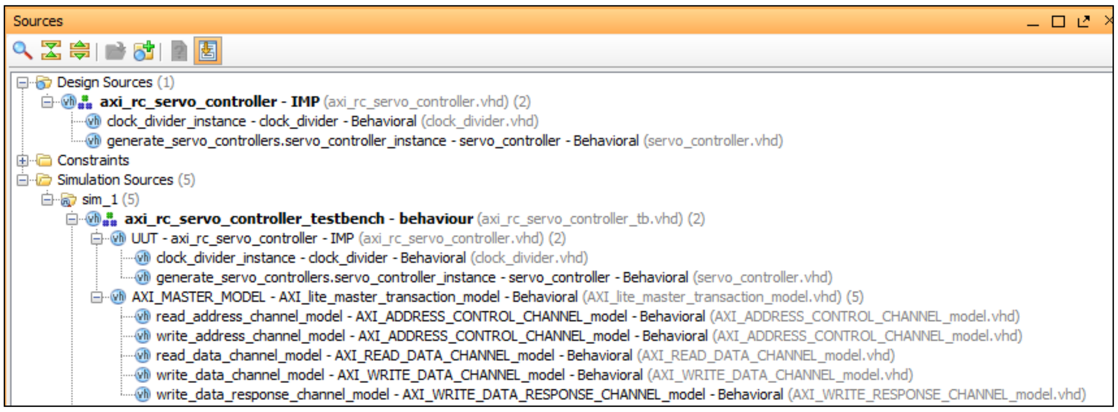
Generating AXI Transactions in the Testbench

With the structured five channel AXI model in place, wrapped by a top level testbench, it is now possible to easily generate AXI transactions in order to test a custom peripheral IP. The following VHDL code shows how easily each AXI transaction can be generated, making testing of the custom peripheral a much simpler task. The two transactions shown here are intended to show both a read and write transaction. The behaviour of the custom peripheral can be observed in the simulation waveforms viewing in the usual fashion, and problems can quickly be found.

```
stimulus : process
begin

-- Set an idle state
address <= X"00000000";
write_data <= X"00000000";
rnw <= '0';
go <= '0';

wait for simulation_interval;


-- Generate a write transaction to the Manual Mode Control Register
address <= X"30000000";
write_data <= X"DEADBEEF";
rnw <= '0';
go <= '1';
wait for AXI_ACLK_period;
wait until done = '1';
go <= '0';
wait for AXI_ACLK_period;
address <= X"00000000";

wait for simulation_interval;

-- Generate a read transaction from the Manual Mode Control Register
address <= X"30000000";
write_data <= X"00000000";
rnw <= '1';
go <= '1';
wait for AXI_ACLK_period;
wait until done = '1';
go <= '0';
wait for AXI_ACLK_period;
address <= X"00000000";

wait for simulation_interval;

{... add additional stimuli here ...}

-- End of Stimuli.  Give some time to finish up.
wait for simulation_interval;

sim_end <= true;
wait;

end process stimulus;
```

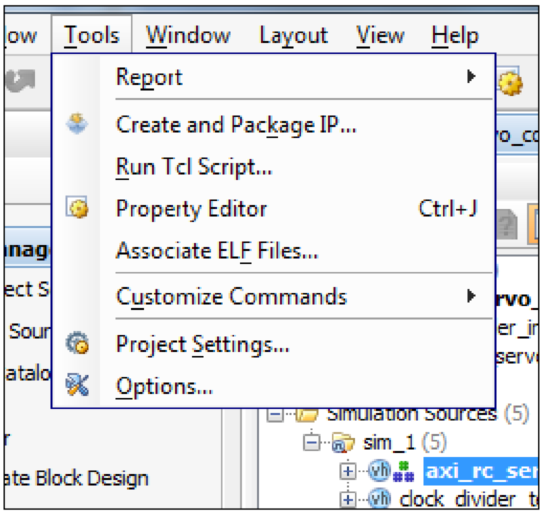
Packaging the IP

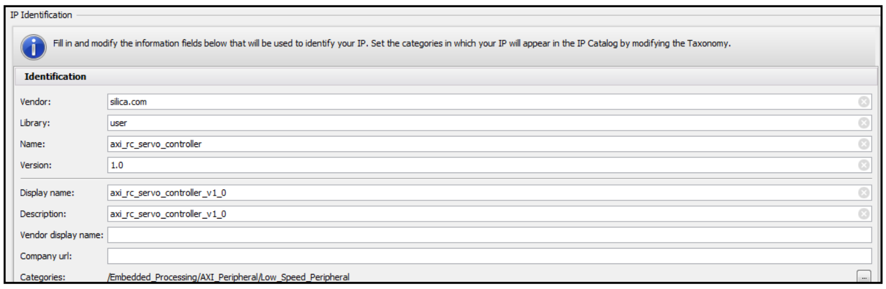
The next step of the design process is to package the IP to put it into a format that can be understood by the Xilinx Vivado block diagram GUI. By this stage you have probably got a set of source files in the Xilinx Vivado source window which look similar to the following screenshot.


Sources

```
Design Sources (1)
  axi_rc_servo_controller - IMP (axi_rc_servo_controller.vhd) (2)
    clock_divider_instance - clock_divider - Behavioral (clock_divider.vhd)
    generate_servo_controllers.servo_controller_instance - servo_controller - Behavioral (servo_controller.vhd)
Constraints
Simulation Sources (5)
  sim_1 (5)
    axi_rc_servo_controller_testbench - behaviour (axi_rc_servo_controller_tb.vhd) (2)
      UUT - axi_rc_servo_controller - IMP (axi_rc_servo_controller.vhd) (2)
        clock_divider_instance - clock_divider - Behavioral (clock_divider.vhd)
        generate_servo_controllers.servo_controller_instance - servo_controller - Behavioral (servo_controller.vhd)
      AXI_MASTER_MODEL - AXI_lite_master_transaction_model - Behavioral (AXI_lite_master_transaction_model.vhd) (5)
        read_address_channel_model - AXI_ADDRESS_CONTROL_CHANNEL_model - Behavioral (AXI_ADDRESS_CONTROL_CHANNEL_model.vhd)
        write_address_channel_model - AXI_ADDRESS_CONTROL_CHANNEL_model - Behavioral (AXI_ADDRESS_CONTROL_CHANNEL_model.vhd)
        read_data_channel_model - AXI_READ_DATA_CHANNEL_model - Behavioral (AXI_READ_DATA_CHANNEL_model.vhd)
        write_data_channel_model - AXI_WRITE_DATA_CHANNEL_model - Behavioral (AXI_WRITE_DATA_CHANNEL_model.vhd)
        write_data_response_channel_model - AXI_WRITE_DATA_RESPONSE_CHANNEL_model - Behavioral (AXI_WRITE_RESPONSE_CHANNEL_model.vhd)
```

In the Vivado tools, when the user is creating an IP the entire Vivado project becomes the level at which the IP can be managed. When the user is ready to package up the IP, the "Create and Package IP" menu item can be chosen from the "Tools" menu at the top of the screen. This menu option starts a tool within the Vivado suite called the "IP Packager", which will take all of the design sources within that project, and start a design wizard which will provide access to all of the configuration settings needed for the IP to be created. The first stage of the wizard asks whether the user wants to create an IP using existing source files, or whether it should create a template for a new IP. This guide details how to create IP from scratch, and therefore the template option is irrelevant for this document. It is usual for the "Package Your Project" option to be chosen at this stage. The location of the output files for the packaged IP are also chosen at this stage, and it is often common practice for a dedicated folder to be established by the user for their own collection of custom IP / peripherals so that a library of IP can be created in the same place. The IP Packager wizard has deceptively few configuration options at this stage, so the user should just click "Finish". When the IP Packager wizard exits, a control file for the packaged IP (component.xml) will be generated in the chosen directory, and automatically added to the Vivado project in a source folder called "IP-XACT". A new screen will also be shown in the Vivado project, and this is where the bulk of the IP configuration takes place. The first tab is called "IP Identification" and allows the user to specify a name for the IP, and set vendor



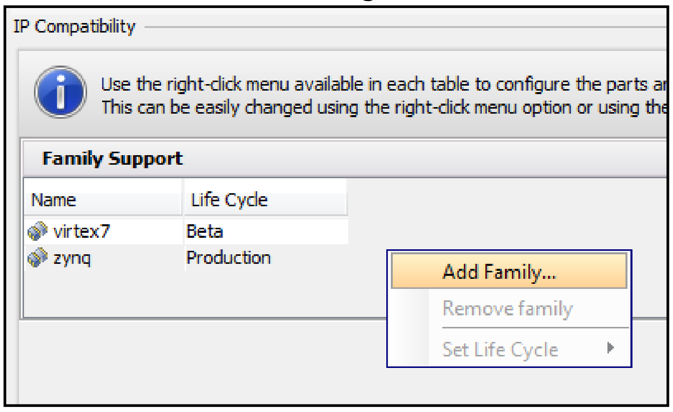

IP Identification

Fill in and modify the information fields below that will be used to identify your IP. Set the categories in which your IP will appear in the IP Catalog by modifying the Taxonomy.

Identification

Vendor:	silica.com
Library:	user
Name:	axi_rc_servo_controller
Version:	1.0
Display name:	axi_rc_servo_controller_v1_0
Description:	axi_rc_servo_controller_v1_0
Vendor display name:	
Company url:	
Categories:	/Embedded_Processing/AXI_Peripheral/Low_Speed_Peripheral

information, version numbers, and a Library name to which the IP belongs. There is also a very important "Category" selection button on this screen which is frequently overlooked, but allows the user to set where the custom IP will appear in the hierarchical IP selection feature within the Block Diagram GUI. For the type of IP being discussed here, the category selection would most likely be "Embedded_Processing → AXI_Peripheral → Low_Speed_Peripheral". The next tab contains the "IP Compatibility" settings. This is a simple way for users to allow their IP to be compatible with different Xilinx device architectures, and to specify which levels of support are available for each architecture. For example, the designer of the IP might have tested it to a production standard for the Zynq architecture, but may have only performed Beta levels of testing on Virtex 7 devices. Furthermore, the IP can offer no support for some other device architectures, for example if certain silicon features are not available. This section of the GUI allows the use to communicate these levels of readiness to end users when the IP is packaged, allowing the user to see only IPs in the catalogue which are suitable for their project. For very specific cases where specialised IP is being developed, it is also possible to lock down the support for specific devices / part numbers, allowing the IP creator to permit usage where only certain devices resources are available. An example of this would be where an IP used a large number of one specific type of resource in the device, such as Block RAMs, DSP slices, or high speed serial transceivers. If only certain members of a device family had sufficient numbers of these resources that were needed by the IP, then the IP designer could specify compatibility with only the suitable members of that device family.

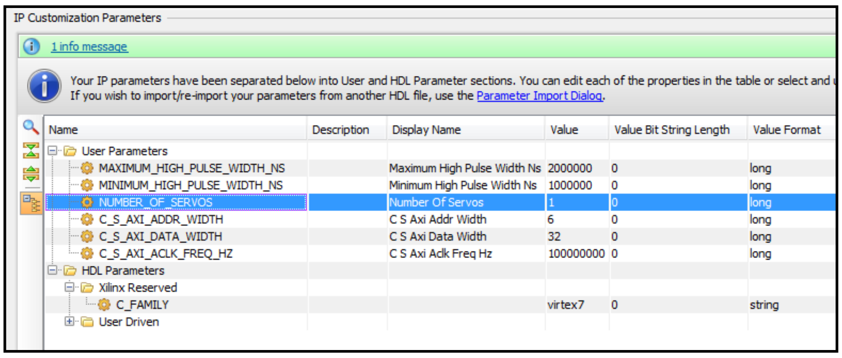

IP Compatibility

Use the right-click menu available in each table to configure the parts a
This can be easily changed using the right-click menu option or using th

Family Support

Name	Life Cycle
virtex7	Beta
zynq	Production

The next tab, "IP File Groups" allows the user to review whether they have correctly assigned each source file to be relevant to the synthesis and / or simulation flows. This is a feature of the Vivado tools which is often overlooked, and when packaging an IP core it is important to ensure that the relevant design sources are added to the synthesis design flows only when they are required. Given that an IP provider is adding files to the design flow of an end user, it is important that testbenches or simulation models should not be added to the synthesis flow of their final design. Common module / entity names such as "adder" and "counter" are frequently used in simulation models, and it would be irresponsible to have untested code synthesised into an end users design flow, simply because a common name was used and the supplied source files were not correctly marked for simulation only. This tab also allows additional directives to be specified for each source file, such as their synthesis order, dependency on other sources. A textual description of each file can also be added here, to enable greater clarity and to assist the end user.

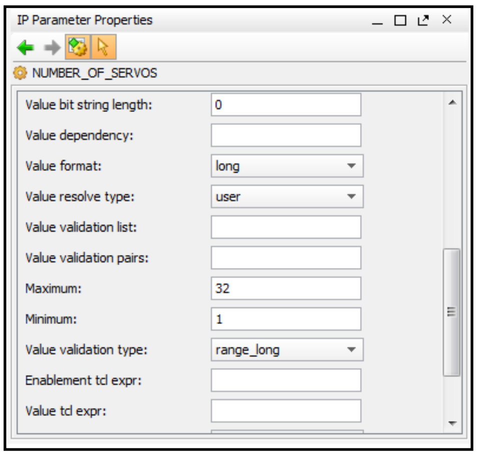


IP Customization Parameters

1 info message

Your IP parameters have been separated below into User and HDL Parameter sections. You can edit each of the properties in the table or select and
If you wish to import/re-import your parameters from another HDL file, use the Parameter Import Dialog.

Name	Description	Display Name	Value	Value Bit String Length	Value Format
User Parameters					
MAXIMUM_HIGH_PULSE_WIDTH_NS		Maximum High Pulse Width Ns	2000000	0	long
MINIMUM_HIGH_PULSE_WIDTH_NS		Minimum High Pulse Width Ns	1000000	0	long
NUMBER_OF_SERVOS		Number Of Servos	1	0	long
C_S_AXI_ADDR_WIDTH		C S Axi Addr Width	6	0	long
C_S_AXI_DATA_WIDTH		C S Axi Data Width	32	0	long
C_S_AXI_ACLK_FREQ_HZ		C S Axi Aclk Freq Hz	100000000	0	long
HDL Parameters					
Xilinx Reserved					
C_FAMILY			virtex7	0	string
User Driven					



IP Parameter Properties

NUMBER_OF_SERVOS

Value bit string length:	0
Value dependency:	
Value format:	long
Value resolve type:	user
Value validation list:	
Value validation pairs:	
Maximum:	32
Minimum:	1
Value validation type:	range_long
Enablement tcl expr:	
Value tcl expr:	

The "IP Customization Parameters" tab allows the user to specify some parameters in the custom IP which can be adjusted from the Vivado tools by the end user. An example of this in VHDL is the use of a Generic on the top level entity declaration. Using parameters, the custom IP can be made flexible and allows the designer to offer the user a much more customizable peripheral. In the example shown here, the parameters allow the user to pass down some timing information which is used by a clock divider and a state machine within the VHDL code of the custom IP. The example also shows a parameter which allows the user to choose the number of output channels on the custom IP, and this is linked to a generate loop statement within the VHDL. To the end user, a vastly different IP can be generated by simply adjusting the number of channels by way of a parameter. The effect of this is that the implemented size of the IP core will vary because the Vivado tools will synthesise the appropriate number of instances of logic buried deep within the peripheral. For each parameter, further information can be added to restrict the values which may be passed by the user to that IP. In the example shown here, the number of channels (servos) is flexible, but constrained to be within 1 and 32. For advanced users, it is also possible to make parameters dependant on each other, and even to implement rules for the setting of parameters based on TCL scripted expressions that can be added to the IP. To aid the user, a "Parameter Import Dialog" is provided as a link at the top of the tab which can be clicked to import the parameters from generics found in the VHDL code. An important parameter to set in this tab is the "C_S_AXI_ADDR_WIDTH" parameter; this tells the Vivado tools how many bits of the address bus are required to decode all of the addressable registers in the IP's address space. In the case of the example, there are more than 32 but less than 64 addressable locations in the IP, and therefore 6 address bits are required ($2^6 = 64$). The importance of this parameter cannot be overstated, because the correct routing of transactions over the AXI interconnect in the rest of the Vivado design depends upon this parameter being set correctly.

The "IP Ports" and "IP Interfaces" tabs are extremely simple, and usually require no adjustment by the user. If, as was suggested at the beginning of this guide, the IP developer has used the recommended port names for the AXI signals in their source code, then the Vivado tools will read the VHDL / Verilog and automatically populate this tab of the IP Packager. If different signal names have been used, the designer will need to identify each of the AXI signals and map them to match the names of the AXI signals that the Vivado tools are expecting.

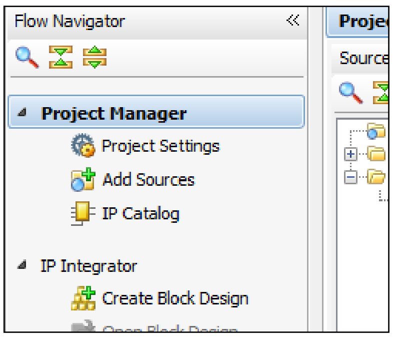
The "IP Addressing and Memory" tab is usually pre-configured by the Vivado tools and should require no user intervention. Assuming that the IP does not have a complex address space (e.g. areas of memory with a break / gap in the middle) then the configuration of this tab is extremely simple and usually automatic. The number of required address bits,

C_S_AXI_ADDR_WIDTH, that was previously specified in the "IP Customization Parameters" tab will be used to create the correct settings by the Vivado tools using a 'raise to the power' equation in the form of "pow(2,(C_S_AXI_ADDR_WIDTH - 1) + 1)".
The penultimate tab, "IP GUI Customization Layout", allows the IP designer to choose how the configuration options for their IP will appear in the Block Diagram editor. This is of particular value when a lot of user configurable parameters have been included in the IP, and allows the designer to create their own layout and presentation for those parameters. Parameters can be grouped together and made to appear on different pages / tabs in the Block Diagram editor. This can promote better usability for the end user of the IP, allowing them to configure the IP more easily. A wizard is provided in this last tab, making the arrangement of the parameters quick and easy. A symbol showing how the IP will appear in the Block Diagram can also be previewed in this tab.

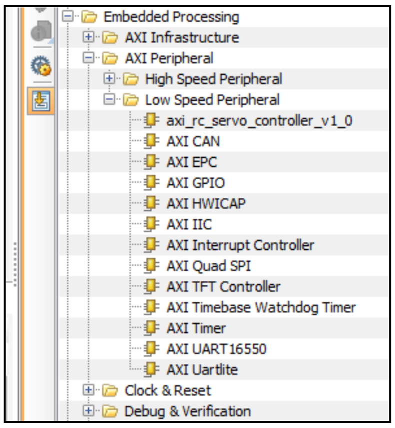
The last tab, "IP licensing and security", allows the user to specify some advanced features relating to IP security and payment based licensing. If the IP is to be distributed and restricted in some way for the user until a payment is received, this tab will allow the relevant features to be added and tie them to a paid licence system. These features are beyond the scope of this guide, but are mentioned here simply for completeness.

When all of the required settings and customizations have been completed, the IP designer should select the "Review and Package" tab. A brief summary of the IP features is shown on this tab, but the main feature is at the bottom of the tab in the form of the "Package IP" button. The IP will then be automatically packaged and a control file generated / updated with the name of "component.xml". This file can be found in the IP-XACT sources folder which will be automatically added to the Vivado project. If changes and modifications to the IP settings are required at a later time, the designer can double-click the XML file, and the IP Packager settings GUI will be re-opened. This completes the packaging process of the IP, making it available to other Vivado projects.

Adding the IP to a Vivado Project

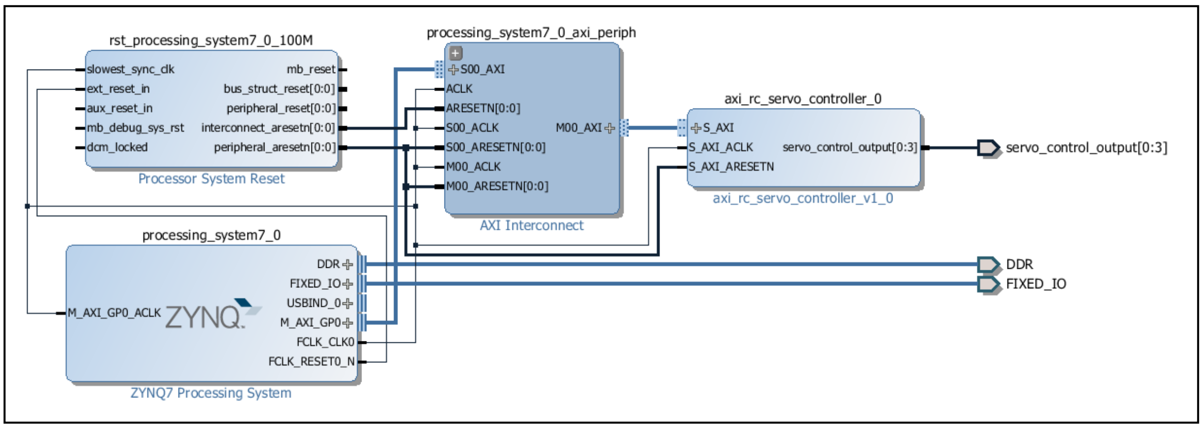


With the packaged IP now complete and ready to be used, a new Vivado project can be created and we can test adding the new IP core to it.
Create a Vivado project, targeting a device architecture that was included in the list of supported architectures in your IP. Open the "IP Catalog" by clicking its link in the Project Manager pane in the top left of the screen. Browse down to the category of IP that you chose when you packaged your IP. In the example shown in this guide, we chose Embedded

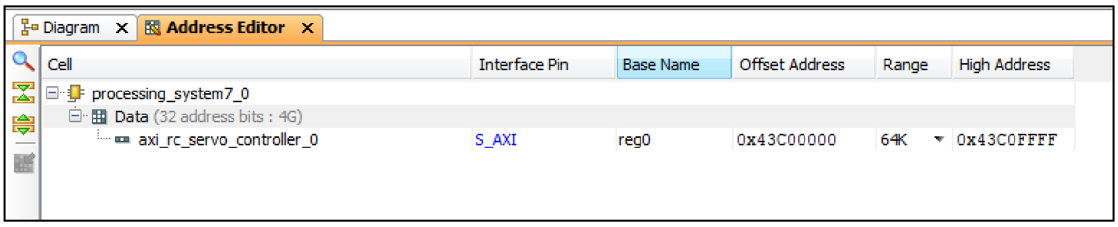


Processing → AXI Peripheral → Low Speed Peripheral. Note that the standard list of low speed IP peripherals are listed, but your custom IP does not appear. This is because we have not yet added your IP project as a repository, so your test Vidado project has no visibility of it at this time. On the left of the list of IPs is a small button for the "IP Settings", represented by some coloured cogs / gears. This button opens a configuration screen and allows the user to define some repositories for user IP. Click the "Add Repository" button and then browse to the location of the project where the custom IP was developed. The previously packaged IP will then be listed in the bottom pane of this configuration screen. Click "OK" and the IP Catalog will now show the custom IP in the Embedded Processing → AXI Peripheral → Low Speed Peripheral folder. Close the IP Catalog, using the "X" icon in the tab title at the top of the pane.

Creating a Block Diagram Including a Custom IP



Click to "Create Block Design" from the "Flow Navigator → IP Integrator" menu on the left of the screen, giving the block diagram a name of your choosing. Add whatever IP you feel is necessary for the integration testing of your custom IP; in our example we will use a Zynq Processing System 7 tile, and an instance of the custom servo controller IP. If you have correctly packaged the custom IP block, the Vivado tools should offer the usual "connection automation" designer assistance features at the top of the block diagram. A block diagram can be quickly assembled, and the Vivado tools will connect all of the AXI connections, clocks, and resets automatically. In our example we have assigned the output of the custom IP to external pins. Be sure to note the base address of the custom IP in the Address Editor tab of the Block Diagram; you will need this later on. In the example shown here, the base address is 0x43C00000.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_rc_servo_controller_0	S_AXI	reg0	0x43C00000	64K	0x43C0FFFF

When the block diagram has been created to match the needs of the IP testing, follow the usual flow in Vivado to create an HDL wrapper, allocate user IO pin constraints, and then synthesise, implement, and create a bitstream for the design. Open the implemented design, and then export the design to the Software Development Kit (SDK) in preparation to write some test software code for the custom IP.

Writing Your First Lines of Test Software Code

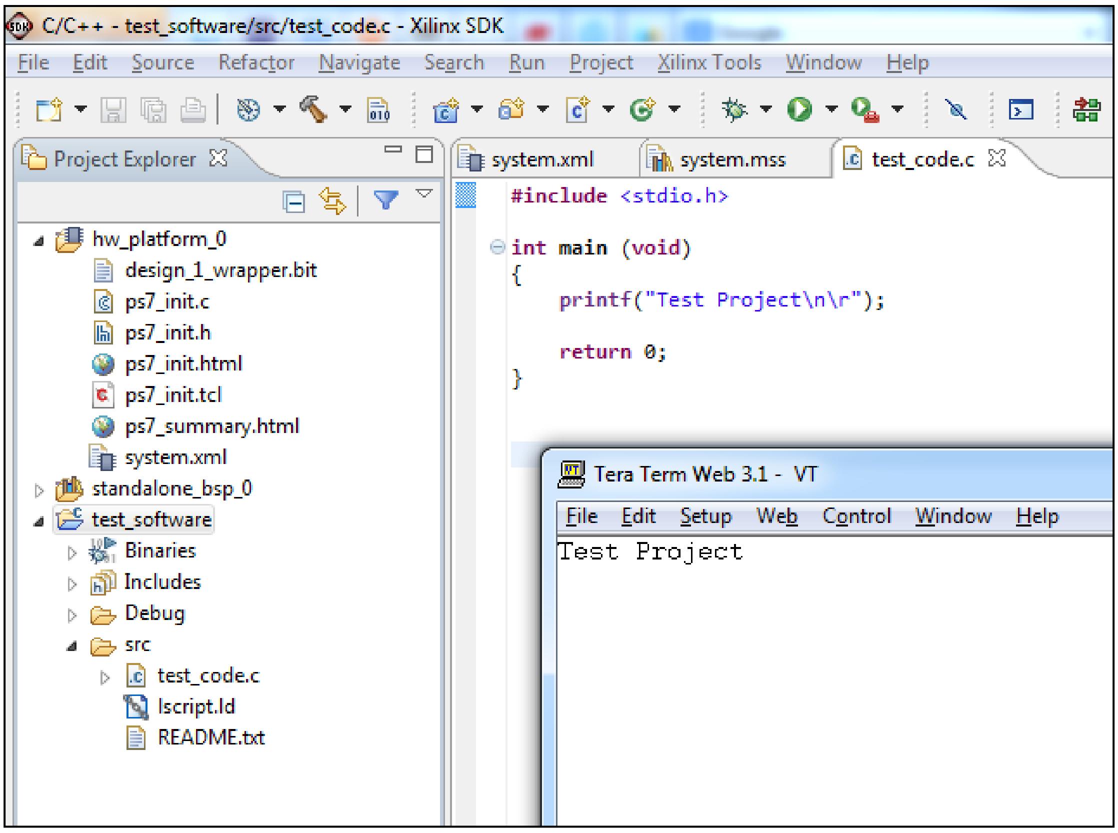
Before writing any test code, it is necessary to create a standalone BSP for your software development. Create this in the usual way using File → New → Board Support Package.

To begin the integration testing of the custom IP, create an software application project using the File → New → Application Project menu item, and choose the "Empty Application" template. Then create a C source file by right-clicking on the "src" folder of your new project and choosing New → Source File, and giving the file a name with a .c extension.

To verify that everything is working properly on the board, start with some simple software code and execute it on the board. In our example we have created a simple application which uses a printf statement to print a line of text to the UART. The purpose of this is simply to prove that the board is executing our software correctly before we continue developing any further code.

```
#include <stdio.h>

int main (void)
{
        printf("Test Project\n\r");

        return 0;
}
```

Create a suitable linker script to link this code to the area of memory that suits your needs, and then run it on the board. Verify that your code is running on the board, remembering to download the bitstream to the board as appropriate.



```
#include <stdio.h>

int main (void)
{
    printf("Test Project\n\r");

    return 0;
}
```

```
Test Project
```

The Beginnings of a Driver

Using the supplied Xilinx IO driver functions is a great way to start when writing your own drivers. The "xil_io" driver provides some useful IO functions which can read and write data values to registers in the programmable logic, and these are perfect when the user wishes to design a driver for a custom IP. Xil_io is easy to use, and can be included in any project using a simple #include statement in the C code. The most commonly used functions calls are Xil_Out32() and Xil_In32(), but similar functions exist for 16 bit and 8 bit data transfers.

Modify your test code to include a write to a register in the custom IP. In the example shown, we are writing a data value of 0xEE to the third 32 bit register in the custom IP's address map, which has an address of 0x43C00008 (base address + 8).

```
#include <stdio.h>
#include "xil_io.h"

int main (void)
{
        printf("Test Project\n\r");

        printf("Writing to a custom IP register...");
        Xil_Out32(0x43C00008, 0xEE);
        printf("Done\n\r");

        return 0;
}
```

It is wise to start your testing by writing to a register that will have a measurable effect on the output of the custom IP. Real life observable feedback is very reassuring during the early stages of testing your own IP. It is much harder to test the functionality of a system where the effect of a modified register cannot easily be observed. Re-run the code on the board and observe the effect. Your results should hopefully match the testing that you performed earlier in simulation.

Writing your drivers with a hierarchical structure is considered to be good design practice, and will save you a lot of time when you need to debug the design. Once you have verified that you have some measurable results by writing to one register, you can then move on to testing some of the others. It is advisable to create a function that can be re-used in your code, because this will form the basis of your driver hierarchy. Modify the code to create two functions; one that writes to the custom IP registers, and one that reads from them.

```
void set_custom_ip_register(int baseaddr, int offset, int value)
        {
        Xil_Out32(baseaddr + offset, value);
        }

int get_custom_ip_register(int baseaddr, int offset)
        {
        int temp = 0;
        temp = Xil_In32(baseaddr + offset);
        return (temp);
        }
```

In these examples, we have implemented the use of offsets from the base address, rather than using hard-coded addresses each time. Another good coding practice is to use #define statements to specify names for commonly used hex values, enabling the user to quickly identify which register is being addressed without having to constantly cross-reference everything to the address map of the IP. The code shown below demonstrates how a simple application can be written that uses the driver functions that we've written. Note the use of the function prototypes, which will be added later to a header file when we move the driver functions to their own source file.

```
#include <stdio.h>
#include "xil_io.h"

#define CUSTOM_IP_BASEADDR 0x43C00000
#define REGISTER_1_OFFSET 0x00
#define REGISTER_2_OFFSET 0x04
#define REGISTER_3_OFFSET 0x08
#define REGISTER_4_OFFSET 0x0C

// Function prototypes
void set_custom_ip_register(int baseaddr, int offset, int value);
int get_custom_ip_register(int baseaddr, int offset);


int main (void)
```

```
{
        int temp3;
        int temp4;

        printf("Test Project\n\r");

        printf("Writing to third register...");
        set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_3_OFFSET, 0xEE);
        printf("Done\n\r");

        printf("Writing to fourth register...");
        set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_4_OFFSET, 0xBB);
        printf("Done\n\r");

        temp3 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_3_OFFSET);
        temp4 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_4_OFFSET);

        printf("Register 3 = 0x%02X\n\r", temp3);
        printf("Register 4 = 0x%02X\n\r", temp4);

        return 0;
}


void set_custom_ip_register(int baseaddr, int offset, int value)
        {
        Xil_Out32(baseaddr + offset, value);
        }

int get_custom_ip_register(int baseaddr, int offset)
        {
        int temp = 0;
        temp = Xil_In32(baseaddr + offset);
        return (temp);
        }
```

The basic functions of our software driver can now be used to create more complex and more useful drivers that will enhance the user's productivity. In our example design, the custom IP controls up to 32 servos. Each servo has a position register, starting at offset 0x08 for servo 1, offset 0x0C for servo 2, and incrementing by four bytes for each servo thereafter. A user friendly function can quickly be written, using the driver functions that we've already written and tested. Numbering schemes can also be corrected in driver functions; for example the user might like to think of servo numbers starting at 1 rather than starting at 0. The following example shows this.

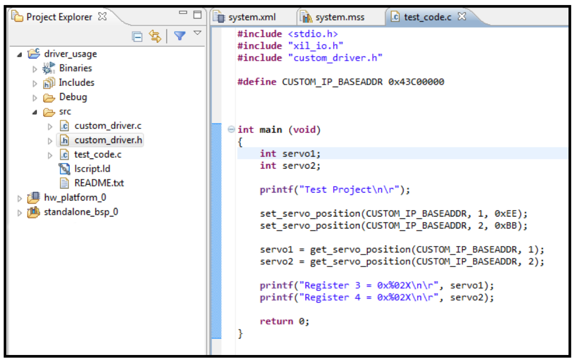
```
void set_servo_position(int baseaddr, int servo_number, int position)
        {
        servo_number--;
        if (servo_number >=0)
                set_custom_IP_register(baseaddr, 8 + (servo_number*4), position);
        }

int get_servo_position(int baseaddr, int servo_number)
        {
        int temp = 0;
        servo_number--;
        if (servo_number >=0)
                temp = get_custom_ip_register(baseaddr, 8 + (servo_number*4));
        return temp;
        }
```

As the driver functions become increasingly advanced, the application code benefits from becoming increasingly simple to read. The following application example shows how a C programmer can set the position of a servo, without requiring any knowledge of the register offsets in the address map. All of the offsets are handled by the driver function, removing the burden from users in all future coding sessions. The application has the same functionality, but is easier to read and manage because we have abstracted all of the detail away into the driver functions.

```c
#include <stdio.h>
#include "xil_io.h"

#define CUSTOM_IP_BASEADDR 0x43C00000

// Function prototypes
void set_custom_ip_register(int baseaddr, int offset, int value);
int get_custom_ip_register(int baseaddr, int offset);
void set_servo_position(int baseaddr, int servo_number, int position);
int get_servo_position(int baseaddr, int servo_number);


int main (void)
{
        int servo1;
        int servo2;

        printf("Test Project\n\r");

        set_servo_position(CUSTOM_IP_BASEADDR, 1, 0xEE);
        set_servo_position(CUSTOM_IP_BASEADDR, 2, 0xBB);

        servo1 = get_servo_position(CUSTOM_IP_BASEADDR, 1);
        servo2 = get_servo_position(CUSTOM_IP_BASEADDR, 2);

        printf("Register 3 = 0x%02X\n\r", servo1);
        printf("Register 4 = 0x%02X\n\r", servo2);

        return 0;
}
```

Moving the Driver Routines to Separate Files

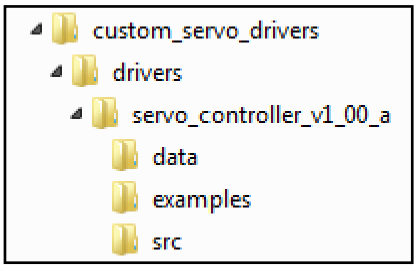
The coding style shown above should illustrate that drivers can quickly be built to exercise the various features of the custom IP. It should also be very obvious that we are starting to generate significant amounts of code, and we already have many functions in our source file which are beginning to get untidy. To continue the development of the driver functions, we can now move the code into a separate file. Specifically, we will move the code for the driver functions into a separate custom_driver.c source file, and the function prototypes into a custom_driver.h header file. To maintain visibility of the functions across the different files, be sure to add the line `#include "custom_driver.h"` at the top of your main application source file, and also to the custom_driver.c source file. With this file structure in place, it is possible to quickly and easily continue the development of the custom IP drivers, while keeping the source code in the main test application tidy and manageable. For very complex driver functions, additional source files can be added to the fileset to facilitate additional hierarchy in the source code.



```c
#include <stdio.h>
#include "xil_io.h"
#include "custom_driver.h"

#define CUSTOM_IP_BASEADDR 0x43C00000


int main (void)
{
    int servo1;
    int servo2;

    printf("Test Project\n\r");

    set_servo_position(CUSTOM_IP_BASEADDR, 1, 0xEE);
    set_servo_position(CUSTOM_IP_BASEADDR, 2, 0xBB);

    servo1 = get_servo_position(CUSTOM_IP_BASEADDR, 1);
    servo2 = get_servo_position(CUSTOM_IP_BASEADDR, 2);

    printf("Register 3 = 0x%02X\n\r", servo1);
    printf("Register 4 = 0x%02X\n\r", servo2);

    return 0;
}
```

Creating the Xilinx Driver File and Folder Structure

When the driver source files have been written and tested, the next step is to put the driver files into a directory tree structure that the Xilinx SDK will understand. This is a very important step because the Xilinx tools will expect to find files and folders with reserved names. An example of the required folder structure is shown here; the top level folder can be called anything that the user chooses, and be placed anywhere that they choose, but must contain a sub-folder called "drivers". Below that, there may be one or more folders which represent each driver for the custom IP. We have just one driver in our example which is called "servo_controller_v1_00_a". The suffix "_v1_00_a" is important is denotes revisions of that driver. The accepted notation is a major/minor numeric revision in decimal digits (1.00 in this example), followed by a sub-version denoted by a single alpha character (in this case "a"). The accepted use of these version numbers is that major and minor changes to the functionality of the driver should be represented numerically, and bug fixes / performance improvements that do not affect the intended functionality should be represented by an alpha character change.



Under the structure described above, three further folders are expected. The "src" folder contains the .c and .h source files for the driver, in addition to a Makefile which can be written to describe the build process required to compile the sources in the correct order / dependency. The "examples" folder contains examples of software application code, showing how your driver might be used in a final application. The "data" folder contains control files which are specific to the operation of the Xilinx SDK tools, and which detail how your various source files should be used. These control files are detailed below.

The MDD file

The first control file in the "data" folder is the Microprocessor Driver Definition (MDD) file. The filename is required to have a suffix of "_v2_1_0.mdd", and in the case of our example has the full filename of "servo_controller_v2_1_0.mdd". The suffix relates to the version of the syntax that is used within the MDD file; in this case v2.10. The contents of the file are relatively simple, and for most MDD files the text is identical with the exception of one or two lines.

```
OPTION psf_version = 2.1;

BEGIN driver rc_servo_controller

  OPTION supported_peripherals = (axi_rc_servo_controller_v2_0);
  OPTION driver_state = ACTIVE;
  OPTION copyfiles = all;
  OPTION VERSION = 3.0;
  OPTION NAME = rc_servo_controller;

END driver
```

Full details of each parameter are available in document UG642 - "Platform Specification Format Reference Manual" (psf_rm.pdf).

The "OPTION supported_peripherals" line should be updated to list the peripherals that are served by the custom driver. In the example shown here there is just one peripheral which will be supported by the driver, but multiple peripherals can be listed, separated by a space. This name must match the name of the IP that you created when using the IP Packager.

The "BEGIN driver" line should be edited to create a name for the driver that is being developed. There are no specific rules for the naming convention of this parameter, but it is wise to create a name that will be clearly identifiable to the end user.

The "OPTION driver_state" line allows the user to maintain a recognised lifespan of their custom driver, and list the driver as either ACTIVE, DEPRECATED, or OBSOLETE as and when the developer choose to supersede or retire it from service. The effect of changing this option away from the "active" state, generates either warnings or errors in the driver compile process when it is invoked by the end user.

The "OPTION depends" line allows the developer to list any dependences of their driver on others. In the case shown in the example here, the "common" drivers would be required before the custom driver was compiled.

The "OPTION copyfiles" line tells the SDK which source files in the custom driver's "src" directory should be copied when the SDK generates the BSP. In most cases this will be left set to "all".

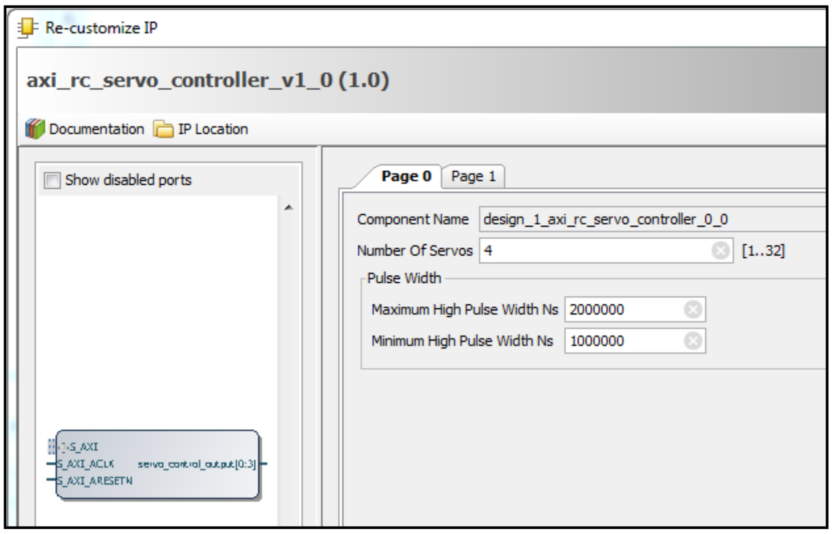
The VERSION line allows the user to specify a version number for their driver. This should match the directory name used previously.

The TCL file

The second control file in the "data" folder is the TCL file, which is used by the Xilinx BSP creation tools to automatically generate some parameters which can be used later by the software developers. The TCL file show below is actually only 4 lines in length, including the final curly bracket on its own line, but lines 2 & 3 are extremely long and therefore difficult to re-produce clearly in this document.

```
proc generate {drv_handle} {
  xdefine_include_file $drv_handle "xparameters.h" "XRCSERVO" "NUM_INSTANCES" "C_BASEADDR" "C_HIGHADDR" "DEVICE_ID" "NUMBER_OF_SERVOS" "MINIMUM_HIGH_PULSE_WIDTH_NS" "MAXIMUM_HIGH_PULSE_WIDTH_NS"
  xdefine_canonical_xpars $drv_handle "xparameters.h" "XRCSERVO" "NUM_INSTANCES" "C_BASEADDR" "C_HIGHADDR" "DEVICE_ID" "NUMBER_OF_SERVOS" "MINIMUM_HIGH_PULSE_WIDTH_NS" "MAXIMUM_HIGH_PULSE_WIDTH_NS"
}
```

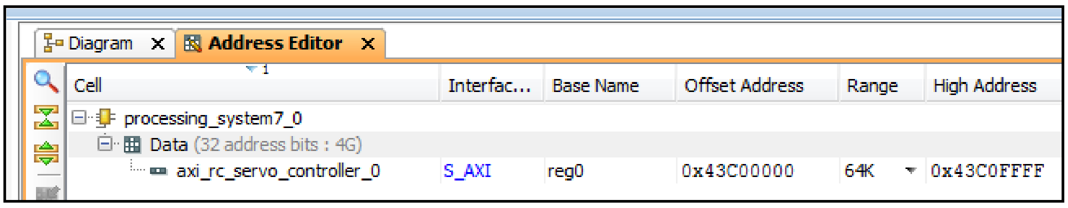
The only lines that should be edited by the developer are lines 2 & 3 (beginning "xdefine_include_file" and "xdefine_canonical_xpars"), and list a number of parameters that will be generated automatically by the BSP generation tools before being placed into a file called "xparameters.h" within the automatically generated board support package. The editable section of this line begins with the parameter after "xparameters.h"; in the case of this example the first two editable parameters are "XSERVO" and "NUM_INSTANCES". These two parameters are used to create a #define statement in the "xparameters.h" file called "XPAR_XSERVO_NUM_INSTANCES" and assign a numerical value to it. The Xilinx BSP generation tools have the ability to automatically count the number of instances of each type of peripheral that are added into the user's embedded processor design. This information can be of great use when the same driver is used to control multiple instances of the same peripheral, and provides the ability for a loop to be created in software that will automatically update when the number of instances of a given peripheral is increased and decreased in the design. In the case of our example there is only one instance of the test peripheral so we need not worry about such a feature, but the line "`#define XPAR_XSERVO_NUM_INSTANCES 1`" will be automatically added to the "xparameters.h" file when the BSP is generated. The list of parameters in the TCL file on the rest of lines 2 & 3 represent additional #define statements that will be generated in the xparameters.h file. Each of the #define statements will have a prefix related to the instance name of the IP, and a suffix copied from the text string in quotes on lines 2 & 3 of the TCL file. Each of the text strings matches the name of a Generic in the top level VHDL entity for the custom IP, and the value of each #define is populated using either the Generic's default value in the VHDL code, or the value that the user has set in the Vivado block diagram tool to override that default. In the case of our example, seven #define statements will be created in the xparameters.h file during BSP generation, and they will be populated according to the user's chosen settings and the number of instances of the custom IP that were added to the user's design. An example of this automated integration is illustrated below:


Re-customize IP

axi_rc_servo_controller_v1_0 (1.0)

Component Name: design_1_axi_rc_servo_controller_0_0
Number Of Servos: 4 [1..32]
Pulse Width
Maximum High Pulse Width Ns: 2000000
Minimum High Pulse Width Ns: 1000000

```
/* Definitions for driver SERVO_CONTROLLER */
#define XPAR_XSERVO_NUM_INSTANCES 1

/* Definitions for peripheral AXI_RC_SERVO_CONTROLLER_0 */
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_BASEADDR 0x43C00000
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_HIGHADDR 0x43C0FFFF
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_DEVICE_ID 0
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_NUMBER_OF_SERVOS 4
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_MINIMUM_HIGH_PULSE_WIDTH_NS 1000000
#define XPAR_AXI_RC_SERVO_CONTROLLER_0_MAXIMUM_HIGH_PULSE_WIDTH_NS 2000000
```



Cell	Interfac...	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_rc_servo_controller_0	S_AXI	reg0	0x43C00000	64K	0x43C0FFFF

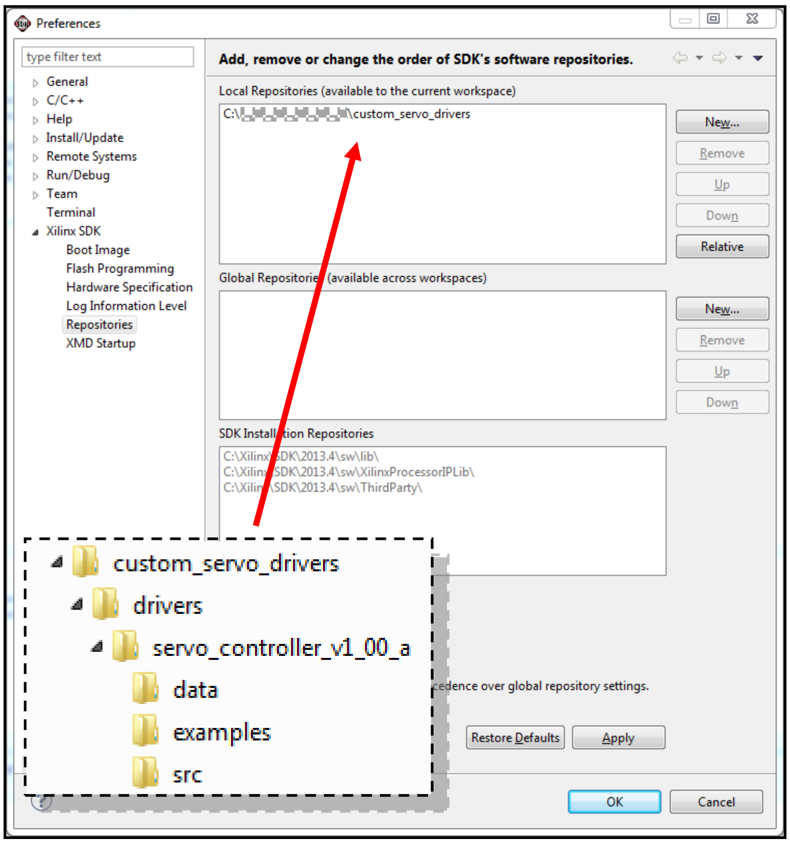
The ability to generate a software BSP which will automatically update itself based upon changes made to the hardware design is an incredibly powerful feature, and can save the software engineering team a lot of manual development effort and time. The end user's software application can then make use of these parameters, allowing the hardware and software teams to work completely independently, yet vastly reduce the possibility for software errors and bugs to occur due to any changes made in the hardware design that were not manually communicated to the software engineering team.

Configuring the Xilinx SDK

With the driver control files written and placed in the correct folder structure, the final step is to configure the Xilinx SDK tools so that they have visibility of the new driver in the list of available driver repositories.

In the SDK, open the "Preferences" dialogue by choosing "Window → Preferences" from the menu bar. Select the "Xilinx SDK → Repositories" pane, as shown in the screenshot.

Add a new repository to the list by clicking the "New..." button next to the "Local Repositories" list, and point to the folder that you created for your custom drivers. The folder you choose here should be the level of the file system above the "drivers" folder.



Add, remove or change the order of SDK's software repositories.

Local Repositories (available to the current workspace)

C:\[redacted]\custom_servo_drivers

Global Repositories (available across workspaces)

SDK Installation Repositories

C:\Xilinx\SDK\2013.4\sw\lib\
C:\Xilinx\SDK\2013.4\sw\XilinxProcessorIPLib\
C:\Xilinx\SDK\2013.4\sw\ThirdParty\

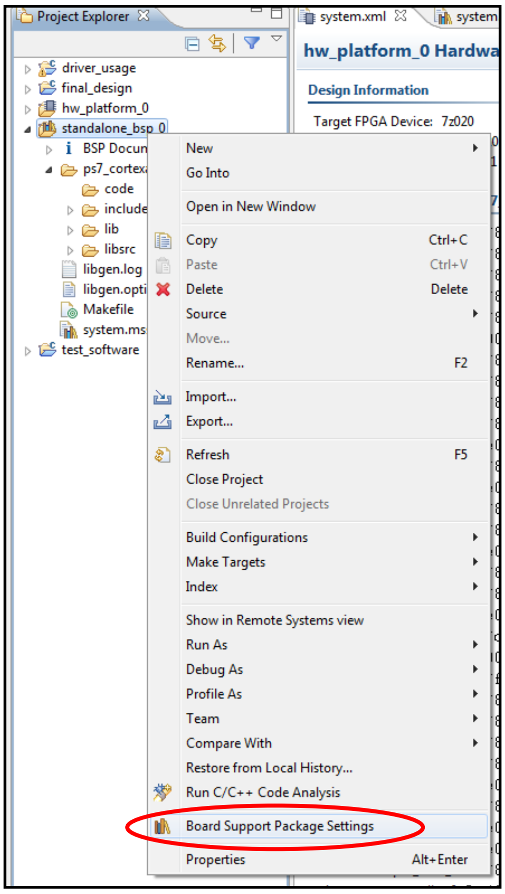
...cedence over global repository settings.

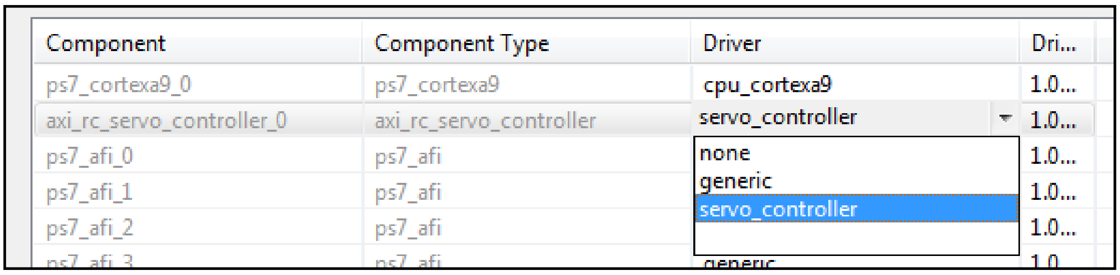
Click the "Rescan Repositories" button, followed by "OK". This setting will provide visibility of your custom driver to the SDK tool, and will enable your driver to be selected in the BSP settings. There is an addition list shown on this screen called "Global Repositories"; this performs an identical function to that of adding the drivers to the "Local Repositories" list, with the exception that the drivers will be visible to any SDK workspace that is created or opened in the future. This is a powerful feature if your goal is to create a single repository of custom drivers for multiple custom IPs, and still have them routinely available and visible to all of your SDK workspaces.

Selecting a Custom Driver in the BSP

The task of creating and configuring your custom IP and custom driver is now complete, and the Vivado and SDK tools will now have visibility of them. The final stage is to select your custom driver and allocate it to be automatically compiled as part of the BSP for your user application.

Right click on the Board Support Package in the Project Explorer that you created earlier (standalone_bsp_0), and choose the "Board Support Package Settings" menu item from the bottom of the list. Choose the "drivers" pane from the choices shown on the left of the window, and then identify the instance of your custom IP in the configuration table shown. It will now be possible to select your custom driver from the drop down menu in the "Driver" column of the table. If you had created multiple folders representing different versions of the same driver, you will also be able to choose the version of the driver in the "Driver Version" column. Click OK and the BSP will automatically be re-generated. If you now examine the "include" and "libsrc" folders in the BSP's source tree, you will find that your header files and C source files have been automatically copied and compiled into the BSP. There is no longer any requirement for custom driver functions to be added to the list of sources for each application, because they are now included in the BSP alongside the drivers provided by Xilinx for supplied IP.





Component	Component Type	Driver	Dri...
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	1.0...
axi_rc_servo_controller_0	axi_rc_servo_controller	servo_controller	1.0...
ps7_afi_0	ps7_afi	none	1.0...
ps7_afi_1	ps7_afi	generic	1.0...
ps7_afi_2	ps7_afi	servo_controller	1.0...
ps7_afi_3	ps7_afi	generic	1.0

Conclusion

That's it! You have now created a custom IP peripheral, added it to a user design, and created custom drivers for the IP. The Xilinx tools allow seamless integration of custom IP into the Vivado and SDK tools, offering the same levels of automation and flexibility for content created outside of the Xilinx catalogue.

Design files are supplied with this document, showing how many of the concepts described can be implemented. Please visit the website to download them.