Summary

UltraScale™ FPGAs feature a built-in primitive named MASTER_JTAG that provides the same functionality as the external JTAG port. The internal FPGA logic can use MASTER_JTAG to load the AES decryption key into the battery-backed RAM (BBRAM). It can also use the MASTER_JTAG primitive to program various one-time programmable (OTP) eFUSEs from within the device, instead of using the external JTAG port. This application note abstracts the low-level commands needed to communicate with the MASTER_JTAG primitive by providing the necessary Xilinx® supplied HDL and C library functions for a MicroBlaze™-based Soft Processor Core [Ref 1] design. These designs can be modified and enhanced for various use cases.

You can download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

Xilinx devices have featured both BBRAM and eFUSE non-volatile (NVM) memory since Virtex®-6 FPGAs. The primary use of a BBRAM in UltraScale devices is to store the 256-bit AES bitstream decryption key. This AES key can also be stored in the eFUSEs with the important difference that it can neither be modified nor erased. A 32-bit USER_EFUSE register that can be programmed for custom use cases is also available.

UltraScale FPGAs have introduced an additional programmable 128-bit register, USER_EFUSE_128, for use as a maintenance log, a tamper activity log, or for a custom use case. A 384-bit eFUSE register that can be programmed with the hash of a RSA-2048 public key as a part of the device’s authentication scheme is also available. The eFUSE-based control bits including read/write disables, decryption disable, external JTAG disable, and RSA enable can also be programmed. Refer to the UltraScale Architecture Configuration User Guide (UG570) [Ref 2] and Vivado Design Suite User Guide: Programming and Debugging (UG908) [Ref 3] for more information.

Traditionally, BBRAM and eFUSEs are programmed through the external JTAG port using the Xilinx ISE iMPACT or Vivado® Hardware Manager. Starting with UltraScale FPGAs, BBRAM and eFUSEs can also be programmed using the internal MASTER_JTAG port.

The ability to program using an internal path has a number of advantages, especially in security applications. For instance, it is possible to perform a field update of the BBRAM key remotely and securely by instantiating a secure key exchange function in the FPGA logic. The new key can travel safely over an unsecured network because it is encrypted (black), decrypted in the FPGA
logic, and then stored in the BBRAM from within the device. Other security use cases are internal programming of eFUSEs such as tamper log information, AES decryptor disable, and JTAG disable as responses to a tamper event. Refer to the Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs Application Note (XAPP1098) [Ref 7].

A MicroBlaze Soft Processor Core [Ref 1] is instantiated in the FPGA logic to interface with the internal MASTER_JTAG port. The Xilinx-supplied XILSKEY library provides an application programming interface (API) to simplify and abstract the necessary low-level JTAG commands. The hardware and software development required to use the XILSKEY library is outlined in Hardware and Software Development Overview. Refer to the LibXil SKey for Zynq-7000 UltraScale and Zynq UltraScale+ MPSoC AP SoC Devices User Guide [Ref 8] for more information about the XILSKEY library.

Following the design flow described in this application note, one Vivado hardware project, and three different SDK applications are created. These SDK applications perform the following tasks:

- Internally program the BBRAM AES key.
- Create a Hello World design with an encrypted bitstream to verify the BBRAM key.
- Internally program eFUSE bits. The internal eFUSE programming is verified by the SDK application as well as by the Vivado Hardware Manager.

This application note assumes that you are familiar with the Vivado tools flow methodology (including IP Integrator—IPI), and are experienced with building simple SDK projects. See Vivado Design Suite User Guide: Programming and Debugging (UG908) [Ref 3], Vivado Design Suite User Guide: Design Flows Overview (UG892) [Ref 4], Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 5], and Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994) [Ref 6] for more information.

---

**Hardware and Software Requirements**

The hardware and software requirements for the internal programming of BBRAM and eFUSEs memory are as follows.

- KCU105 Kintex® UltraScale FPGA evaluation kit [Ref 9] based on the XCKU040-2FFVA1156E FPGA
- AC to DC power adapter (12 VDC)
- USB type-A to micro-B USB cable for JTAG
- USB type-A to micro-B USB cable for UART
- Xilinx Vivado design suite version 2016.3 or later [Ref 10]
- Xilinx Software Development Kit (XSDK) version 2016.3 or later [Ref 11]
- Serial communications terminal application such as Tera Term [Ref 12]
Hardware and Software Development Overview

**IMPORTANT:** The BBRAM key cannot be used if the EFUSE_KEY_ONLY control eFUSE is already programmed on the KCU105 evaluation board’s FPGA. Refer to the UltraScale Architecture Configuration User Guide (UG570) [Ref 2] for more information.

**Special Considerations When Using MASTER_JTAG to Internally Program eFUSEs**

Certain special considerations must be taken to ensure reliability when MASTER_JTAG is used to internally program eFUSEs.

1. The configuration memory must not be accessed during internal programming. This includes activities such as a single-event upset (SEU) scan, readback capture, and partial reconfiguration (PR). For example, the Xilinx Soft Error Mitigation (SEM) IP core [Ref 13] should not be operational during eFUSE programming.

2. The board/system design must take into consideration the additional current required for programming eFUSEs. The programming current is available in the appropriate UltraScale device data sheets [Ref 14].

**Hardware and Software Development Overview**

This application note outlines the steps to create a MicroBlaze processor-based Vivado tools design that indirectly interfaces the MASTER_JTAG primitive using the general purpose I/O (GPIO). A Vivado hardware project is created and exported to the SDK. The SDK tool is launched with the MicroBlaze processor-based Vivado tools design as its hardware platform. Three different software applications based on the available software libraries and example code are then created. After these SDK applications are compiled and built into executable and linkable format (ELF) files, they are brought back into the Vivado tools. Three separate FPGA bitstreams are then generated to perform and verify the functions described in this application note.

In addition to the MicroBlaze processor, two VHDL files are included in this project (see Reference Design for more information):

- int_prog_ctl_us.vhd
- jtag_monitor.vhd

**INT_PROG_CTL_US Module**

Most of the time the JTAG signaling can come directly from the MicroBlaze processor’s GPIO interface. However, to provide precise control over the duration of the actual eFUSE programming pulse, the INT_PROG_CTL_US module is required. This module instantiates the MASTER_JTAG primitive.

The INT_PROG_CTL_US module passes the MicroBlaze processor’s GPIO signals directly to and from the MASTER_JTAG primitive, except when it enters the actual eFUSE programming phase. Upon entering this phase, the XILSKEY software polls for the INT_PROG_CTL_US module using a READY signal. After the software detects that the module is READY, it asserts the START_PROG input to command the module to take control of the JTAG signaling (TCK and TMS) to get a
precise 5 µs programming pulse. When the eFUSE programming is complete, the INT_PROG_CTL_US module asserts the END_PROG signal and allows the software to regain control of the JTAG signals. This interface is illustrated in Figure 1.

If you want to internally program just the BBRA M and not the eFUSE, then you can ignore this module. Xilinx recommends including this module because it consumes few resources and allows future internal eFUSE programming. However, if you decide to ignore it, then you must instantiate the MASTER_JTAG primitive and connect it directly to the MicroBlaze processor’s GPIO.

If your design is also instantiating the Xilinx Security Monitor IP Core (see the Security Monitor IP Core Product Brief [Ref 15]), the INT_PROG_CTL_US module must be modified, because the Security Monitor instantiates the MASTER_JTAG primitive. The MASTER_JTAG primitive must be removed from the INT_PROG_CTL_US module and the corresponding JTAG signals brought up to the module’s port list to connect to the Security Monitor’s MASTER_JTAG interface.

**JTAG_MONITOR Module**

The optional JTAG_MONITOR module provides an EFUSE_PROG_PULSE output that can be routed up to an external header pin on the KCU105 to measure the eFUSE programming pulse width (this signal mirrors the amount of time the JTAG controller is in the programming phase).

The JTAG_MONITOR can also be used to debug/monitor the JTAG signals and display the current JTAG state. When using the Vivado Integrated Logic Analyzer (ILA) IP core to monitor the JTAG states, the INT_PROG_CTL_US module must have its TEST_MODE generic set to true (MASTER_JTAG is not instantiated) to allow the ILA to communicate using external JTAG. Otherwise, the ILA does not work, because external JTAG is disconnected when MASTER_JTAG is instantiated within a design.

The JTAG_MONITOR module is useful for development and debugging. It is optional and can be removed in a field deployed system.

**Building the Vivado Hardware Project**

One common Vivado project is the hardware required for this application note. It is based upon a MicroBlaze soft processor using GPIO to indirectly interface with the MASTER_JTAG primitive through the INT_PROG_CTL_US Module. The following steps are used to create the Vivado hardware project.
1. Open the Vivado tools (version 2016.3 or later) in graphical user interface (GUI) mode.
2. From the welcome screen under Quick Start select **Open Example Project** (Figure 2).
3. Click **Next**, select **Base MicroBlaze**, and click **Next** (Figure 3).

![Vivado Example Project](image)

**Figure 2:** Vivado Example Project
4. Enter the project name: **internal_prog_proj**.

5. Enter the appropriate project location and (optionally) keep the create project subdirectory checkbox selected.

   **Note:** The maximum allowed path length on Windows is 260 bytes.

6. Click **Next**, select **Kintex-UltraScale KCU105 Evaluation Platform**, and click **Next** (Figure 4).
7. Click **Finish**. The Vivado tools create the base MicroBlaze processor project (**Figure 5** and **Figure 6**).
8. Double-click the **Clocking Wizard** IP block, select the **Output Clocks** tab, verify **clk_out1** is selected, and the requested and actual frequency is 100 MHz. Click **OK**. This is the main clock for the MicroBlaze core.
9. Under the Output Clock column, check the box next to `clk_out2` and type in 8.000. Verify that the actual column reports back 8.000 MHz. This is the clock used by the `INT_PROG_CTL_US Module` to provide precise time during the actual programming of the eFUSEs. Click **OK** (Figure 7).

**Figure 7: Clocking Wizard**

**IMPORTANT:** For eFUSE programming, the 8.000 MHz clock feeds into the `INT_PROG_CTL_US block`. This is critical for generating the required programming pulses. The JTAG TAP machine must transition using an accurate clock during this phase to provide a 5 µs pulse for each eFUSE programmed. If the programming pulses are not accurate, eFUSE programming and reliability can be adversely affected.

10. Right-click on the `clk_out2` port of the **Clocking Wizard** IP block, select **Create Port**, set **Interface Name** to `CLK_8MHZ`, and click **OK** (Figure 8).
11. From within the IPI block diagram window search and add the following IP blocks by selecting the **Add IP** icon on the left side of the IPI window.

   a. **System Management Wizard** (used to monitor on-chip temperature and voltage prior to reading and writing eFUSEs).

   b. **AXI Timer** (used as a watchdog in the eFUSE programming code section).

12. Towards the top of the IPI window, next to where it says Designer Assistance available, click **Run Connection Automation**.

13. Select the **All Automation** checkbox to select all the checkboxes and click **OK** to automatically connect up the recently added IP blocks (**Figure 9**).

---

**Figure 8: Create Port**

![Create Port](image)
14. Right-click on the **External Interface**: `led_8bits` connected to the AXI GPIO IP block and click **Delete**.

15. Double-click the **AXI GPIO** IP block to re-customize the IP.

16. Using the drop-down boxes, ensure that both the IP Interfaces GPIO and GPIO2 are set to **Custom Board Interface** (Figure 10).
17. Select the IP Configuration tab, check the All Outputs checkbox for GPIO, set GPIO Width to 4, check the Enable Dual Channel checkbox, check the All Inputs checkbox for GPIO 2, set GPIO Width to 3, and click OK (Figure 11).
18. Right-click on the GPIO port of the AXI GPIO IP block, select Create Interface Port, set the Interface Name to GPO, and click OK (Figure 12).

19. Right-click on the GPIO2 port of the AXI GPIO IP block, select Create Interface Port, set the Interface Name to GPI, and click OK (Figure 13).
20. Right-click on the **Vp_Vn** port of the System Management Wizard IP block and select **Make External**.

21. Select the **Address Editor** tab at the top of the IPI window, set the **Range** for **microblaze_0_local_memory/dlmd_bram_if_cntlr** to **256K** using the drop-down box, and set the **Range** for **microblaze_0_local_memory/ilmd_bram_if_cntlr** to **256K** using the drop-down box (Figure 14).

22. Select the **Diagram** tab at the top of the IPI window to return to the block diagram.

23. Click the **Validate Design** icon on the left side of the IPI window and correct any issues if the validation does not pass.

24. The large text labels on the block diagram can be moved and/or renamed.

25. Click the **Regenerate Layout** icon on the left side of the IPI window to clean up the block diagram display. It should look similar to Figure 15. After the edits are complete, save the block diagram.
26. From the top-level Vivado tools menu, select Tools > Project Settings > Target Language VHDL > OK.

27. In the center window, select the Sources tab, select the Hierarchy tab, right-click on base_mb_i, select Create HDL Wrapper, choose Copy generated wrapper to allow user edits, and click OK (Figure 16 and Figure 17).

Figure 15: MicroBlaze-based Internal BBRAM/eFUSE Programming Project Layout
28. In the center window, under Design Sources, double-click `base_mb_wrapper` to bring up the HDL file in the text editor and make the following changes:

   a. Remove the GPIO ports from the top-level port list and assign as internal signals.

   ![Figure 16: Sources](image)

   ![Figure 17: Create HDL Wrapper](image)
b. Instantiate and connect the INT_PROG_CTL_US component (this block instantiates the MASTER_JTAG primitive). See the INT_PROG_CTL_US Module for more information.

c. Instantiate and connect the JTAG_MONITOR component (optional). See the JTAG_MONITOR Module for more information.

29. The following VHDL code sample demonstrates these changes (the VHDL wrapper file can also be downloaded from Xilinx.com). See Reference Design for more information.

```vhdl
entity base_mb_wrapper is
  port (
    Vp_Vn_v_n : in  std_logic;
    Vp_Vn_v_p : in  std_logic;
    default_sysclk_300_clk_n : in  std_logic;
    default_sysclk_300_clk_p : in  std_logic;
    reset : in  std_logic;
    rs232_uart_rxd : in  std_logic;
    rs232_uart_txd : out std_logic;
    EFUSE_PROG_PULSE : out std_logic
  );
end base_mb_wrapper;

architecture STRUCTURE of base_mb_wrapper is
  signal clk_8mhz : std_logic;
  signal gpi_tri_i : std_logic_vector(2 downto 0);
  signal gpo_tri_o : std_logic_vector(3 downto 0);
  signal jtag_mon : std_logic_vector(3 downto 0);

  component base_mb is
    port (
      default_sysclk_300_clk_n : in  std_logic;
      default_sysclk_300_clk_p : in  std_logic;
      rs232_uart_rxd : in  std_logic;
      rs232_uart_txd : out std_logic;
      Vp_Vn_v_n : in  std_logic;
      Vp_Vn_v_p : in  std_logic;
      reset : in  std_logic;
      CLK_8MHZ : out std_logic;
      GPI_tri_i : in  std_logic_vector(2 downto 0);
      GPO_tri_o : out std_logic_vector(3 downto 0)
    );
  end component base_mb;

  component INT_PROG_CTL_US is
    generic (
      TEST_MODE : boolean := FALSE
    );
    port (
      RESET : in  std_logic;
      CLK_8MHZ : in std_logic; -- accuracy of the 5µs efuse prog pulse depends on this 8MHz clock
      GPIO_IN : in  std_logic_vector(3 downto 0); -- from software using gpio output
      GPIO_OUT : out std_logic_vector(2 downto 0); -- to software using gpio input
      JTAG_OUT : out std_logic_vector(3 downto 0)  -- for monitoring jtag signals
    );
  end component INT_PROG_CTL_US;

  component JTAG_MONITOR is
    port ( JTAG_MON : in  std_logic_vector(3 downto 0); -- tdi=0, tms=1, tck=2, tdo=3
      EFUSE_PROG_PULSE : out std_logic
    );
  end component JTAG_MONITOR;
```
begin

base_mb_i: base_mb
   port map (
      CLK_8MHZ => clk_8mhz,
      GPI_tri_i => GPI_tri_i,
      GPO_tri_o => GPO_tri_o,
      Vp_Vn_v_n => Vp_Vn_v_n,
      Vp_Vn_v_p => Vp_Vn_v_p,
      default_sysclk_300_clk_n => default_sysclk_300_clk_n,
      default_sysclk_300_clk_p => default_sysclk_300_clk_p,
      reset => reset,
      rs232_uart_rxd => rs232_uart_rxd,
      rs232_uart_txd => rs232_uart_txd
   );

INT_PROG_CTL_US_i: INT_PROG_CTL_US
   generic map (
      TEST_MODE => FALSE
   )
   port map (
      RESET => reset,
      CLK_8MHZ => clk_8mhz,
      GPIO_IN => GPO_tri_o,
      GPIO_OUT => GPI_tri_i,
      JTAG_OUT => jtag_mon
   );

JTAG_MONITOR_i: JTAG_MONITOR
   port map (
      JTAG_MON => jtag_mon,
      EFUSE_PROG_PULSE => EFUSE_PROG_PULSE
   );

end STRUCTURE; -- base_mb_wrapper

30. Download the project VHDL files int_prog_ctl_us.vhd and jtag_monitor.vhd to your local computer. See Reference Design for more information.

31. In the center Sources window, right-click on Design Sources, select Add Sources, click Next, select Add or create design sources, and click Add Files. Navigate to and add int_prog_ctl_us.vhd and jtag_monitor.vhd (files downloaded in step 30), check Copy Sources into project, and click Finish (Figure 18).
32. Download the project constraint file xapp1283.xdc to your local computer (see Reference Design for more information). The constraint file sets up the external reset pin input and the EFUSE_PROG_PULSE monitor output pin.

33. In the center Sources window, right-click on Design Sources, select Add Sources, select Add or create constraints, click Next, and click Add Files. Navigate to and add xapp1283.xdc (file downloaded in the step 32), check Copy constraints files into project, and click Finish (Figure 19).
34. From the left Flow Navigator window, select Run Implementation under the Implementation section, click Yes to automatically launch synthesis prior to implementation.

**Note:** Any critical synthesis or implementation warnings associated with the board’s dip switch can be safely ignored because it is not being used in this application note.

35. After the implementation is successfully completed, click Cancel.

36. From the top-level Vivado tools menu, select File, select Export, choose Export Hardware, and leave Include bitstream unchecked. Click OK to export Local to Project (Figure 20).
37. From the top-level Vivado tools menu, select **File > Launch SDK**. Click **OK** to launch **Local to Project** (Figure 21).

![Launch SDK](image)

*Figure 21: Launch SDK*

**Building the SDK Applications**

The method to create three different SDK applications is explained in this section. In step 37 of the Vivado hardware section the SDK tool is launched with the MicroBlaze processor design as its hardware platform. The following steps are required for the creation of the SDK software applications.

1. From the top-level SDK menu, select **File > New > Board Support Package**.
2. Set the project name to **internal_prog_bsp**, check the **Use default location** checkbox, select **standalone Board Support Package OS**, and click **Finish** (Figure 22).
3. From the Board Support Package Settings popup window, select **xilskey 6.0** (or later) **Xilinx Secure Key Library** and click **OK** (Figure 23).
4. Select the `system.mss` tab in the center window and then select **Import Examples** under the **Libraries** section at the bottom.

5. Check the `xilskey_bbram_ultrascale_example` checkbox, check the `xilskey_efuse_example` checkbox, and click **OK** (Figure 24).
6. On the left side of the SDK project explorer window, expand the `internal_prog_bsp_xilskey_bbram_ultrascale_example_1` project, expand `src`, and double-click `xilskey_bbram_ultrascale_input.h` to open up the header file in the SDK text editor.

7. Scroll down the header file to find the `#define statement for `XSK_BBRAM_AES_KEY` and note and record the key value in quotations. Save any changes. The default value can be used in this example or you can enter your own value using exactly 64 hexadecimal characters. For this project, the obfuscated key (XSK_BBRAM_OBFUSCATED_KEY) is not used and can be ignored (Figure 25).

```
/*
 * This is the 256 bit key to be programmed into BBRAM.
 * This should be entered by user in HEX.
 */

// hack
#define XSK_BBRAM_OBFUSCATED_KEY "b1c276899d71fb4cdd4a9a7985ea46c2e11f9374d09c7ea23b7eb67de713cfd1"
#define XSK_BBRAM_AES_KEY "0e66e900000000005241583950bceda1eadcdeabcdece43216615aaabbbabab"
```

Figure 25: **XSK_BBRAM_AES_KEY Value**

8. On the left side of the SDK project explorer window, expand the `internal_prog_xilskey_efuse_example_1` project, expand `src`, and double-click `xilskey_input.h` to open up the header file in the SDK text editor.

9. Scroll down the header file to find the `#define statement for `XSK_EFUSEPS_DRIVER` and comment it out (this is only used for Zynq® SoC designs). Save the changes (Figure 26).
10. Scroll through the `xilskey_input.h` file to see the different options (commands and values) for programming eFUSEs. For programming details refer to LibXil SKey for Zynq-7000 UltraScale and Zynq UltraScale+ MPSoC AP SoC Devices User Guide [Ref 8]. In this example a value is programmed into the 32-bit USER_EFUSE register.

**IMPORTANT:** eFUSEs are OTP and the programming is irreversible. Be extremely careful in choosing the eFUSEs that are programmed because they can affect the configuration and operation of the device.

**IMPORTANT:** Do not enable XSK_EFUSEPL_ENABLE_RSA_AUTH (PL RSA authentication eFUSE bit) for the XCKU040-2FFVA1156E FPGA device located on the KCU105 evaluation platform. UltraScale devices have a minimum configuration width when RSA is enabled and the KCU105 evaluation board does not conform to this requirement. Enabling the PL RSA authentication eFUSE bit makes the device on this board permanently unconfigurable.

11. Find the `#define` statement for XSK_EFUSEPL_PROGRAM_USER_KEY and set its value to TRUE to program the 32-bit USER_EFUSE register. Save the changes (Figure 27).

```
#define XSK_EFUSEPL_PROGRAM_USER_KEY TRUE /**< TRUE burns
    * the USER key */
```

**Figure 27:** XSK_EFUSEPL_PROGRAM_USER_KEY Value

12. Find the `#define` statement for XSK_EFUSEPL_READ_USER_KEY and set its value to TRUE in order to read the 32-bit USER_EFUSE register. Save the changes (Figure 28).

```
#define XSK_EFUSEPL_READ_USER_KEY TRUE /**< TRUE read
    * USER key */
```

**Figure 28:** XSK_EFUSEPL_READ_USER_KEY Value

13. Find the `#define` statement for 32-bit USER_EFUSE register value XSK_EFUSEPL_USER_KEY (this is not the AES decryption key) and set its value using exactly eight hexadecimal characters. All hexadecimal combinations are permitted with the exception of 0x00000000. Note and record this value. Save the changes (see the example value in Figure 29).

```
#define XSK_EFUSEPL_USER_KEY "012345ED"
```

**Figure 29:** XSK_EFUSEPL_USER_KEY Value
14. From the top-level SDK menu, select **File**, select **New**, select **Application Project**, and set the project name to **hello_encrypted**. Under Board Support Package, check **Use existing**, click **Next**, select **Hello World** template, and click **Finish** (Figure 30 and Figure 31).

*Figure 30: Create an Application Project*
From the top-level SDK menu, select Project Clean > Clean all projects, and click OK.

From the top-level SDK menu, select Project and click Build All. This generates the necessary binary ELF files for each of the three application projects.

Bitstream Generation

In this section, the three ELF files created in the SDK are imported back into the Vivado tools to merge the associated MicroBlaze processor instructions and data into each of the bitstreams. The bitstreams for programming the BBRAM AES key and the 32-bit user eFUSE register are unencrypted and the Hello application is encrypted (using the same key value from the C header file). The following steps outline the generation of these bitstreams.

1. Return to the Vivado tools GUI with the recently built internal_prog_proj project loaded.
2. In the center window, select the Sources tab and then select the Hierarchy tab.
3. Right-click on Design Sources > Add Sources > Add or create design sources > Next > Add Files.
4. Browse to <project_path>/internal_prog_proj/internal_prog_proj.sdk. From this location point to each of the following ELF files and add them to the project:

   a. hello_encrypted/<Debug or Release>/hello_encrypted.elf
   b. internal_prog_xilskey_bbram_ultrascale_example_1/<Debug or Release>/
      internal_prog_bsp_xilskey_bbram_ultrascale_example_1.elf
   c. internal_prog_xilskey_efuse_example_1/<Debug or Release>/
      internal_prog_bsp_xilskey Efuse_example_1.elf

5. Do not check Copy sources into project.

6. Click Finish (Figure 32).

7. The three ELF files should now be visible under Design Sources and ELF (Figure 33).
8. Under Design Sources right-click on `internal_prog_bsp_xilskey_bbram_ultrascale_example_1.elf` > Associate ELF Files. Under Associated ELF File for Design Sources browse, select `internal_prog_bsp_xilskey_bbram_ultrascale_example_1.elf`, click OK, and click OK (Figure 34).

9. From the flow navigator window on the left of the Vivado tools screen, under Program and Debug select **Generate Bitstream**.
10. When the bitstream generation is completed, click **Cancel**.

11. Using a file browser, navigate to **project_path>/ internal_prog_proj/internal_prog_proj.runs/impl_1** (this path is used again when the bitstreams are programmed into the device).

12. Rename **base_mb_wrapper.bit** to **prog_bbram.bit**.

13. Under Design Sources right-click on **internal_prog_bsp_xilkeys_efuse_example_1.elf**, select **Associate ELF Files**. Under Associated ELF File for Design Sources, browse **...**, select **internal_prog_bsp_xilkeys_efuse_example_1.elf**, click **OK**, and click **OK** (Figure 35).

14. From the flow navigator window on the left of the Vivado tools screen, under Program and Debug, select **Generate Bitstream**.

15. When the bitstream generation is completed, click **Cancel**.

16. Using a file browser, navigate to **<project_path>/ internal_prog_proj/internal_prog_proj.runs/impl_1**.

17. Rename **base_mb_wrapper.bit** to **prog_efuse.bit**.

18. Under Design Sources, right-click on **hello_encrypted.elf > Associate ELF Files**.
   Under Associated ELF File for Design Sources, browse **...**, select **hello_encrypted.elf**, click **OK**, and click **OK** (Figure 36).
19. From the flow navigator window on the left of the Vivado tools screen, under Implementation, select **Open Implemented Design**.

20. Ignore any warning messages that appear and click **OK**.

21. From the flow navigator window on the left of the Vivado tools screen, under Program and Debug, select **Bitstream Settings**.

22. Select **Configure additional bitstream settings**, on the left select **Encryption**, set **Enable Bitstream Encryption: Yes**, and **Select location of encryption key: BBRAM**.

23. Copy and paste the 256-bit AES key (the XSK_BBRAM_AES_KEY value from xilskey_bbram_ultrascale_input.h) from Building the SDK Applications section into the **Starting AES encryption key (key0)** text box.

24. Set **Number of encryption blocks per key: 256**, click **OK**, and click **OK** (Figure 37). For more information on encryption block selection, refer to the UltraScale Architecture Configuration User Guide (UG570) [Ref 2].
25. From the flow navigator window on the left of the Vivado tools screen, under Program and Debug, select **Generate Bitstream**. Generate Bitstream, select **Save**, check **Select an existing file**, set the filename as `xapp1283.xdc`, click **OK**, and click **Overwrite** (Figure 38 and Figure 39). This adds the encryption options and key value to the constraints file.
26. When the bitstream generation is completed, click **Cancel**.

27. Using a file browser, navigate to
   
   `<project_path>/internal_prog_proj/internal_prog_proj.runs/impl_1`

28. Rename `base_mb_wrapper.bit` to `hello_encrypted.bit`.

**Verification**

After the three bitstreams `prog_bbram.bit`, `prog_efuse.bit`, and `hello_encrypted.bit` are successfully generated, you can load each onto the Kintex UltraScale device located on the KCU105 evaluation kit and verify their functionality. The following steps outline the verification of each of these bitstreams.

1. Power up and connect the KCU105 evaluation board to a computer using the USB JTAG and USB UART cables. Refer to the *KCU105 Quick Start Guide* (XTP391) [Ref 16] and *KCU105 Board User Guide* (UG917) [Ref 17] for more information.

2. On the KCU105 evaluation board set dip switch **SW15** to the following settings for the JTAG configuration mode:
   
   - Positions 1 through 5 = **OFF**
   - Position 6 = **ON**

3. Determine the COM port that connects to the MicroBlaze processor's UART using Windows device manager. The USB UART connection establishes two COM ports (standard and enhanced) on the computer (the Silicon Labs serial driver must be installed on the system). The computer COM port associated with the Silicon Labs Standard COM port connects to the MicroBlaze processor's UART.

4. Start up the serial communications terminal program. The serial port settings should be as follows:
   
   - COM Port = Silicon Labs Standard COM Port
   - Baud rate: 115200
   - Data: 8
   - Parity: none
   - Stop: none
   - Flow control: none

**Figure 39: set_property**
5. Press and release the **PROGRAM_B** switch **SW4** on the KCU105 board.

**IMPORTANT:** The **PROGRAM_B** switch must be pressed between each bitstream load because the design contains the **MASTER_JTAG** primitive. Failure to do so blocks the external JTAG path and the computer’s USB JTAG communications fail.

6. In the Vivado tools from the flow navigator window on the left of the screen, under Program and Debug, select **Hardware Manager**, select **Open Target**, and click **Auto Connect**.

7. In the center hardware window, right-click on the **xcku040_0** device, select **Clear BBR Key**, and click **OK** (Figure 40).

8. In the center hardware window, right-click on the **xcku040_0** device, select **Program Device**, set **Bitstream file: <path_to_bitstreams>/hello_encrypted.bit**, and click **Program** (Figure 41).
9. Set **Bitstream file: <path_to_bitstreams>/hello_encrypted.bit** and click **Program** (Figure 42).

10. The hardware manager attempts to program the device. However, because there is no BBRAM key, the configuration fails. When the bitstream finishes loading, verify that the DONE LED (DS34) is OFF.

11. Power cycle the board with the main power switch **SW1**. Then press and release the **PROGRAM_B** switch **SW4** on the KCU105 board.

12. In the center hardware window, right-click on the **xcku040_0** device, select **Program Device**, set **Bitstream file: <path_to_bitstreams>/prog_bbram.bit**, and click **Program** (Figure 43).
13. When the bitstream finishes loading, the DONE LED (DS34) turns ON. Because the MASTER _JTAG is instantiated in the design, it blocks the FPGA’s external JTAG port and causes communication errors in the Vivado Tcl console. These errors are expected and can be safely ignored.

14. The serial terminal window should display a message for a successfully programmed and verified BBRAM (Figure 44).

15. Press and release the PROGRAM_B switch SW4 on the KCU105 board.

16. In the center hardware window, right-click on the xcku040_0 device, select Program Device, set Bitstream file: <path_to_bitstreams>/hello_encrypted.bit, and click Program.

17. When the bitstream finishes loading, the DONE LED (DS34) turns ON because the BBRAM key is loaded internally by the last bitstream (ignore the errors in the Vivado Tcl console).

18. The serial terminal window should display a Hello World message that is appended to the last message. This indicates that the prog_bbram.bit bitstream (loaded earlier), wrote the BBRAM key successfully (Figure 45).
19. Press and release the **PROGRAM_B** switch **SW4** on the KCU105 board.

20. Clear the serial terminal program screen.

21. If you instantiated the optional **JTAG_MONITOR Module**, connect an oscilloscope probe to J53-2 (PACKAGE_PIN AM16). This is the EFUSE_PROG_PULSE signal that mirrors the actual eFUSE programming phase. You should see a 5 µs pulse for each eFUSE bit that is programmed from a 0 to a 1.

22. In the center hardware window, right-click on the **xcku040_0** device, select **Program Device**, set **Bitstream file: <path_to_bitstreams>/prog_efuse.bit**, and click **Program**.

   **Note:** The **prog_efuse.bit** file is not encrypted. It can be encrypted (if required) using the BBRAM key because the key is already loaded in the device.

23. When the bitstream finishes loading, the DONE LED (DS34) turns ON (ignore the errors in the Vivado Tcl console).

24. The serial terminal window should display a message for a successfully programmed eFUSE. The 32-bit USER_EFUSE register should contain the value that you entered earlier in the header file for XSK_EFUSEPL_USER_KEY (Figure 46).
25. The eFuse can be further verified using the Vivado Hardware Manager.

26. Press and release the **PROGRAM_B** switch **SW4** on the KCU105 board.

27. In the center hardware window, right-click on the **xcku040_0** device > **Refresh Device**.

28. In the center hardware window, right-click on the **xcku040_0** device > **Hardware Device Properties**.

29. In the hardware device properties window (directly below the hardware window), expand the properties tree at **REGISTER** and then **EFUSE**. The FUSE_USER field should contain the recently programmed eFuse value (**Figure 47**).

---

**Figure 46:**  Programmed eFuse

**Figure 47:**  FUSE_USER
Conclusion

This application note describes a procedure to internally program the BBRAM and eFUSE memory of an UltraScale FPGA device. Internal programmability is valuable, especially in the context of security-critical applications. The method presented takes advantage of the MASTER_JTAG primitive, an existing MicroBlaze-based hardware example design, Xilinx-supplied HDL modules, a software library (XILSKEY), and software examples that are used to create functional applications. The procedure explained in this application note can modify and enhance examples to fit various use models including a tamper event driven model to impose penalties and a field update of a BBRAM AES key over a secure link.

Reference Design

You can download the reference design files for this application note from the Xilinx website. Table 1 shows the reference design matrix.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
</tr>
<tr>
<td>Developer name</td>
<td>Ed Peterson</td>
</tr>
<tr>
<td>Target devices</td>
<td>UltraScale FPGAs</td>
</tr>
<tr>
<td>Source code provided</td>
<td>Yes</td>
</tr>
<tr>
<td>Source code format</td>
<td>VHDL</td>
</tr>
<tr>
<td>Design uses code and IP from existing Xilinx application note and reference designs or third party</td>
<td>No</td>
</tr>
<tr>
<td><strong>Simulation</strong></td>
<td></td>
</tr>
<tr>
<td>Functional simulation performed</td>
<td>No</td>
</tr>
<tr>
<td>Timing simulation performed</td>
<td>No</td>
</tr>
<tr>
<td>Test bench used for functional and timing simulations</td>
<td>No (Verification using ILA debugger)</td>
</tr>
<tr>
<td>Test bench format</td>
<td>N/A</td>
</tr>
<tr>
<td>Simulator software/version used</td>
<td>N/A</td>
</tr>
<tr>
<td>SPICE/IBIS simulations</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td></td>
</tr>
<tr>
<td>Synthesis software tools/versions used</td>
<td>Vivado design suite 2016.3</td>
</tr>
<tr>
<td>Implementation software tools/versions used</td>
<td>Vivado design suite 2016.3</td>
</tr>
<tr>
<td>Static timing analysis performed</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### References

1. MicroBlaze Soft Processor Core
2. UltraScale Architecture Configuration User Guide (UG570)
7. Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs Application Note (XAPP1098)
8. LibXil SKey for Zynq-7000 UltraScale and Zynq UltraScale+ MPSoC AP SoC Devices User Guide (UG643)
9. Kintex Ultrascale FPGA KCU105 Evaluation Kit
10. Vivado Design Suite
11. Xilinx Software Development Kit
13. Soft Error Mitigation Core
14. UltraScale device data sheets:
   - UltraScale Architecture and Products Overview (DS890)
   - Kintex UltraScale Architecture Data Sheet: DC and AC Switching Characteristics (DS892)
   - Virtex UltraScale Architecture Data Sheet: DC and AC Switching Characteristics (DS893)
15. Security Monitor IP Core Product Brief
16. KCU105 Quick Start Guide (XTP391)
17. KCU105 Board User Guide (UG917)
Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/08/2017</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>

Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at [http://www.xilinx.com/legal.htm#tos](http://www.xilinx.com/legal.htm#tos); IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at [http://www.xilinx.com/legal.htm#tos](http://www.xilinx.com/legal.htm#tos).

Automotive Applications Disclaimer

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2017 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. ARM ™ is a ® trademark of ARM in the EU and other countries. All other trademarks are the property of their respective owners.