EPCS based RSU example on the BeMicro SDK

September 2011
ACDS 11.0sp1
What is RSU?

Here’s the description of Remote System Update from the ALTREMOOTE_UPDATE users guide:

The ALTREMOOTE_UPDATE megafucntion implements a remote system upgrade (also known as remote system update) by taking advantage of the dedicated remote system upgrade circuitry available in supported devices.

A remote system upgrade helps you deliver feature enhancements and bug fixes without recalling the product, and reduces time-to-market and extends product life. By using the ALTREMOOTE_UPDATE megafucntion and the dedicated circuitry, your design can download a new configuration image from a remote location, store it in the configuration memory, and direct the dedicated remote system upgrade circuitry to start a reconfiguration cycle.

The dedicated circuitry performs error detection during and after the configuration process. If there are errors, the circuitry facilitates system recovery by reverting back to a safe, default factory, configuration image and then provides error status information.

In other words:

Remote System Update (RSU) is a capability that many Altera FPGA device families offer thru the ALTREMOOTE_UPDATE megacore which allows the user design in the FPGA to interact with the built in configuration controller that performs Active Serial or Active Parallel configuration of the FPGA. With this internal access to the configuration controller, a user design can direct the configuration controller load a new user design from an arbitrary flash address that contains another configuration image. There are various mechanisms provided for the user design to deal with errors that may occur during this process.
Basic RSU

At power up, the FPGA configuration controller will configure the FPGA with the default “factory boot” image out of the flash in AS or AP configuration mode. The ALTREMOTE_UPDATE block provides status information that allows this initial factory boot image to understand how it got into the FPGA, this might be the initial boot load immediately following power up, or this may be a subsequent reload following the reconfiguration to an application image. If this is a subsequent reload of the factory boot image it may be because the application image that we reconfigured to has decide to reconfigure back to the factory boot image, or maybe there was an error during the reconfiguration attempt into the application image. Anyway the details of the current status of the configuration controller are visible to the factory boot image, and it can decide what it wants to do next, it may do nothing, or it may decide to locate a viable application configuration image stored in the flash device and trigger a reconfiguration into that application image.

Once the application image is loaded it may run indefinitely, and it also has access to the configuration controller. The application image may choose to reconfigure the FPGA back into the factory boot image when ever it wishes. If an error occurs during the configuration into the application image, the configuration controller will automatically reload the factory boot image.
Basic RSU Diagram

- Factory Boot Configuration
  - Power Up
  - Configuration Error
  - Set control registers and reconfigure
  - Application 1 Configuration
    - Reload a different application
  - Application N Configuration
    - Reload a different application
    - Set control registers and reconfigure
    - Configuration Error
RSU error exceptions

Under normal operation, the factory boot hardware image is only loaded into the FPGA under two conditions, power up of the device, or an application image that intentionally chooses to reconfigure back into the factory image. Any other reason that the factory boot image is loaded into the FPGA would be due to an error or exception that occurs during the application reconfiguration event.

The Cyclone 4 RSU core provides the factory boot image with 5 status bits which define why the factory boot image was loaded into the FPGA:

1 – RUNCONFIG – this bit indicates that an application image intentionally reconfigured back into the factory boot image.

2 – WDTIMER – this bit indicates that an application image which was loaded into the FPGA failed to properly maintain the RSU watchdog timer which was enabled when the application image was configured into the FPGA. The RSU watchdog timer is optionally enabled by the factory boot image when it triggers a reconfig.

3 – NSTATUS – this bit indicates that an error was encountered as an application image bit stream was being loaded into the FPGA or the nstatus pin of the FPGA was asserted external to the device.

4 – CRCERROR – this bit indicates that a crc error was encountered in the application image bit stream as it was being loaded into the device.

5 – NCONFIG – this bit indicates that the nconfig pin of the FPGA was asserted while the application image was loaded in the FPGA.

NOTE: At initial power on, none of these bits are set in the status register.
The RSU functionality inside the FPGA is pretty trivial, your design writes the address of a new configuration that you’d like to run and then writes a register to initiate the configuration. That rather simple piece of hardware logic is generally the easy part of a remote system update process. The harder part of the problem is figuring out how to transport new update images into the FPGA configuration flash and organizing the flash in such a way that your FPGA design can figure out what options it has available to reconfigure into.

This example demonstrates one way of implementing a robust method of for remote system update, it is a high availability and fault tolerant flow. This example deploys on the BeMicro SDK evaluation board. This board has a Cyclone IV EP4CE22 FPGA on it which gets configured from an EPCS16 flash device in Active Serial configuration mode. This example uses a Nios II processor running from onchip RAM and a basic compliment of peripherals to implement the remote system update flow. The Nios II processor is programmed to accept new update images for both software and hardware which it programs into the EPCS flash and it can then be commanded to restart the software program or trigger a hardware reconfiguration using the RSU logic in the FPGA. It can also automatically make this decision if it sees credible data in the flash as the factory boot software application executes.
The basic strategy.

At power up, an initial factory boot configuration is loaded into the FPGA from the EPCS flash. The Nios II processor in this design executes a boot loader stored in onchip ROM which locates a factory boot image application out in the EPCS flash, and it loads that software application into onchip RAM and executes it. That factory boot software application investigates the contents of the EPCS flash to see if there are any valid hardware application images available to reconfigure into. This example allows for 2 hardware application images to be stored in the EPCS flash along with the factory boot image, so that once a valid hardware application image is programmed into flash, it can be left there undisturbed as future hardware and software image downloads are attempted. If a new hardware application image is downloaded and stored in flash successfully, then that new hardware image will become the desired application image to be loaded by the factory boot image. In this way the 2 allocated storage locations will ping pong back and forth as new updates are stored in the location that currently does not contain the latest hardware image.

The same strategy is used for software application images as well. There are 2 storage allocations defined to hold application image software applications, that way any valid hardware application image may download new software updates that do not affect the hardware image, they just update the software which runs on the latest valid hardware image.
The Nios II system that we build for both our factory boot image and our application images looks like the Qsys system below. Most of the components in this system are there to make the Nios II software applications more interesting for demonstration purposes. A system that is intended to simply deal with RSU management could be much simpler.

There is a Nios II/f processor that boots from a 4K boot ROM and a 2K boot RAM. It has a 32K and 16K RAM that it can load a program from EPCS flash into and execute. There is a sysid so we can identify the system. There is a jtag uart for the software application console and a separate download jtag uart that we use to simulate an image update download channel. There is a sys timer that we use for alarm services and a wakeup timer that allows us to monitor our boot times. A couple of PIO peripherals allow us to set the state of some LEDs and monitor the state of the user push button on the board. There is an EPCS flash controller that allows us to communicate with the external flash device, and the remote update controller that allows us to interact with the FPGA configuration controller.
Custom RSU Component

All of the components in our Qsys system are standard Altera components except the remote update controller, it is a custom Qsys component that we created. Altera provides a Qsys remote update controller component for Cyclone III, but not for any other device families. Below you see the files that comprise our custom component, you can locate this component directory in the “ip” directory contained in the Quartus project directory for the example project. The directory rsu_cyclone4 contains our component. The file altremote_update_cyclone4.v came from the altremote_update megawizard, configured for Cyclone 4. The rsu_cyclone4.v was taken from the Cyclone III Remote Update Controller component and edited to instantiate the Cyclone 4 altremote_update module that we just created. The rsu_cyclone4_hw.tcl script was created by the component editor when we imported this new component into Qsys. The example project contains software, rsu.h and rsu.c which interacts with this component.
The Nios II system communicates with the external EPCS flash via the EPCS flash controller in the Qsys system. The Nios II can set the state of the LEDs and it can observe the state of the user push button. The reset push button is used to reset the Qsys system, and the RECFG push button can reconfigure the FPGA.
The first choice that we make is how we want to locate objects in the EPCS flash device. The FPGA configuration controller dictates that the factory boot image must reside at location 0x00 in the flash. It turns out that a hardware configuration image for our design is consuming just under 256KB and since the EPCS16 flash sectors are 64KB in size, we decide to allocate 4 sectors to hold each of our potential hardware images, one factory boot image, and two application images. And since our software applications are never going to exceed 64KB, we just allocate one flash sector for each of those, one for our factory boot application software application and one sector each for the two application software applications that we wish to store.
Flash allocation thoughts.

Keep in mind that the flash allocation that was chosen for this example is completely arbitrary. The only specific requirement was the location of the factory boot image, we cannot change that since after power up this is the default location that the configuration controller will load from. The only other requirement that we have for application hardware images is that they begin on a 4 byte boundary. This requirement is driven by the RSU controller which provides an address register for us that contains the upper 22 bits of a 24 bit address to fetch the next configuration from. So application images must be 4 byte aligned in the flash.

Now the reality of flash memory is that when you erase flash, you must erase an entire sector, so for practical reasons, it makes sense to offset our application images such that they start on flash sector boundaries, that way we can safely erase an older image without jeopardizing the valid images that may currently be stored.

This same logic applies to our software application images as well. We should allocate those such that they start on flash sector boundaries as well.
Flash image headers

The general rule that our factory boot software application will follow is to load the application image from flash that represents the latest version that is valid. The way that we deal with this is by appending a header to the beginning of the hardware or software image before we download it and store it in the EPCS flash. This header contains a signature field that identifies it as a hardware or software image header. It also contains a field to hold the sysID ID and TS values from the Qsys system in the hardware image, and we also use these to associate our software applications with a given hardware image. There is a version field that tells us what hardware version or software version is contained in this image. Then we place the length of the image and a crc for the image in the header. Hardware images have a watchdog timeout field that is used to configure the RSU watchdog timer when this image is chosen to be configured. And finally there is a crc field for the header itself, this allows the software application to quickly validate the header as it queries the objects in flash.

When we download a new image, we capture the header, and then download the image and program it into flash. After the entire image is programmed, we validate the image in flash against the crc that we captured in the header. If the image validates against the header then we program the header into the flash right in front of the image, just like we downloaded it. This way if we later validate the header in flash, we can reasonably assume that the image is also valid. If we can’t validate the image after we download it then we don’t bother programming the header into the flash, which would indicate an invalid image.

```c
#define SW_HDR_SIGNATURE ( 0xAA55A5A5 )
#define HW_HDR_SIGNATURE ( 0x55AA5A5A )

//
//  Image header as a structure.
//
struct image_header_s {
    alt_u32 signature;
    alt_u32 hw_id;
    alt_u32 hw_ts;
    alt_u32 version;
    alt_u32 image_length;
    alt_u32 image_crc;
    alt_u32 res_wd_tmo;
    alt_u32 header_crc;
};
```
Image creation

If you study the source materials provided with this example you’ll see that we provide a couple of shell scripts that package our hardware and software images for us to download as updates to the target. These scripts leverage various other utilities supplied in the Altera tools installation.

The hardware image creation starts by using SOF2FLASH to extract the configuration bit stream data from the hardware SOF file. The software image creation starts by using ELF2FLASH to extract the program data from the software ELF file. The extracted data in both cases is then manipulated to create the header to append in front of the image data. Once we get the binary data for the header and image catenated together, we encode the image as base64 for downloading into the target as a remote update image. This final encoding into base64 was an arbitrary choice that we made for this example to facilitate download into the target.

The factory boot hardware image and the factory boot software image are processed with similar utilities but are left in SREC format so they can be programmed into flash with the nios2-flash-programmer. These images represent the factory boot images that you would likely have programmed into your flash at the time of manufacture. You might actually wish to program in an initial application image as well at manufacture, but that level of detail is not a concern of this example.
What does our example do?

A step by step walk thru.
When you download the example project source package and build it, the build flow creates a factory boot hardware image and accompanying software application as well as 3 hardware application images each with 3 software images. You then run the demo walk through script, and at this point it doesn’t matter what is running in the FPGA or stored in the flash. This demo will erase all of those contents and replace them with it’s own.
Erasing the flash

To begin with the example wants a completely blank flash device, to accomplish this it uses nios2-configure-sof to download the factory boot hardware image into the FPGA so that it can run nios2-flash-programmer to erase the flash.
Next we use the nios2-flash-programmer to program the factory boot hardware image into flash.
And the factory boot software

Then we use the nios2-flash-programmer again to program the factor boot software image into flash.
Now that we have the factory boot hardware and software images programmed into flash, we can press the RECFG push button on the board, which removes the current FPGA contents and replaces them with the factory boot image from the EPCS flash.
Once the FPGA is configured with the factory hardware image, the Nios II begins running from it’s reset location which happens to be the onchip ROM and RAM. This is a boot loader program attempts to locate a factory boot software application in flash that can be loaded and executed in the onchip RAM.
Application load

The factory boot image boot loader copies the factory boot software application from the flash into the onchip RAM and then jumps to its entry point.
At this point the factory boot software application runs on the Nios II processor. It will query the flash objects looking for a valid application image to reconfigure the FPGA with. When it finds nothing available in the flash, this application jumps into a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.
From the host system, we download the first application hardware image into the flash via the download JTAG UART.
Then we download the first software application image for the application hardware image into the flash via the download JTAG UART.
At this point now that we have a valid application hardware image and a software image for it to run, we send a hardware reconfiguration command down to the target over the download JTAG UART.
Upon receiving the hardware reconfiguration request over the download JTAG UART the software application scans the flash for a valid application image and software application. When it finds one, it programs the RSU controller with the appropriate address information and triggers the reconfiguration.
The reconfiguration controller now loads the application image into the FPGA.
Once the FPGA is configured with the application hardware image, the Nios II begins running from its reset location which happens to be the onchip ROM and RAM. This boot loader program attempts to locate an application software application in flash that can be loaded and executed in the onchip RAM.
The application image boot loader copies the application software application from the flash into the onchip RAM and then jumps to its entry point.
At this point the application software application runs on the Nios II processor. It will start a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.

Software application run

Cyclone IV FPGA

Qsys System

Nios II/f

2KB Boot RAM

32KB RAM

sysid

4KB Boot ROM

16KB RAM

JTAG UART

sys Timer

RSU Controller

download JTAG UART

wakeup Timer

EPCS Flash Controller

Application Software 1

Factory Boot Software

Application 1 Image

Factory Boot Image

RECFG PB

Reset PB

User PB

8 LEDs
Now we download a software update for the application hardware image into the flash via the download JTAG UART.

Software image download

Cyclone IV FPGA

Qsys System

- Nios II/f
- sysid
- JTAG UART
- download JTAG UART

2KB Boot RAM
4KB Boot ROM
sys Timer
wakeup Timer

32KB RAM
16KB RAM
RSU Controller
pb PIO
led PIO

EPCS Flash

- Application Software 2
- Application Software 1
- Factory Boot Software
- Application 1 Image
- Factory Boot Image

RECFG PB
Reset PB
User PB
8 LEDs
Trigger a software restart

At this point now that we have a new software image for this hardware application image to run, we send a software restart command down to the target over the download JTAG UART.
Application boot loader

The old application software application closes down the hardware and jumps back thru its reset vector. The boot loader program attempts to locate an application software application in flash that can be loaded and executed in the onchip RAM. This time it locates the new software application.
The application image boot loader copies the application software application from the flash into the onchip RAM and then jumps to its entry point.
At this point the application software application runs on the Nios II processor. It will start a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.
Now we download another software update for the application hardware image into the flash via the download JTAG UART.
Trigger a software restart

At this point now that we have a new software image for this hardware application image to run, we send a software restart command down to the target over the download JTAG UART.
The old application software application closes down the hardware and jumps back thru its reset vector. The boot loader program attempts to locate an application software application in flash that can be loaded and executed in the onchip RAM. This time it locates the new software application.
The application image boot loader copies the application software application from the flash into the on-chip RAM and then jumps to its entry point.
At this point the application software application runs on the Nios II processor. It will start a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.
Now we download a new hardware application image update into the flash via the download JTAG UART.
Then we download an application software application that can run on our new hardware application image into the flash via the download jtag uart.
At this point now that we have a new valid application hardware image and a software image for it to run, we send a hardware reconfiguration command down to the target over the download jtag uart.
Trigger a hardware reconfig

Upon receiving the hardware reconfiguration request over the download jtag uart the software application triggers the reconfiguration. There is only one place the application image can reconfigure to, the factory boot image, so there is no elaborate algorithm required to figure this out, we just write the reconfiguration register.
Reconfiguration

The reconfiguration controller now loads the factory boot image into the FPGA.
Factory boot loader

Once the FPGA is configured with the factory hardware image, the Nios II begins running from its reset location which happens to be the onchip ROM and RAM. The boot loader program attempts to locate a factory boot software application in flash that can be loaded and executed in the onchip RAM.
Application load

The factory boot image boot loader copies the factory boot software application from the flash into the onchip RAM and then jumps to its entry point.
At this point the factory boot software application runs on the Nios II processor. It will query the flash objects looking for a valid application image to reconfigure the FPGA with. It locates the latest valid hardware application image and it elects to trigger a reconfiguration.
Trigger a hardware reconfig

The Nios II programs the RSU controller with the appropriate address information and triggers the reconfiguration.
Reconfiguration

The reconfiguration controller now loads the application image into the FPGA.
Once the FPGA is configured with the application hardware image, the Nios II begins running from its reset location which happens to be the onchip ROM and RAM. This boot loader program attempts to locate an application software application in flash that can be loaded and executed in the onchip RAM.
The application image boot loader copies the application software application from the flash into the onchip RAM and then jumps to its entry point.
Software application run

At this point the application software application runs on the Nios II processor. It will start a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.
Now we download another hardware update for the application hardware image into the flash via the download JTAG UART.
Then we download another software update for the application hardware image into the flash via the download JTAG UART.
At this point now that we have a new valid application hardware image and a software image for it to run, we send a hardware reconfiguration command down to the target over the download JTAG UART.
Upon receiving the hardware reconfiguration request over the download JTAG UART, the software application triggers the reconfiguration. There is only one place the application image can reconfigure to, the factory boot image, so there is no elaborate algorithm required to figure this out; we just write the reconfiguration register.
Reconfiguration

The reconfiguration controller now loads the factory boot image into the FPGA.
Factory boot loader

Once the FPGA is configured with the factory hardware image, the Nios II begins running from it’s reset location which happens to be the onchip ROM and RAM. The boot loader program attempts to locate a factory boot software application in flash that can be loaded and executed in the onchip RAM.
The factory boot image boot loader copies the factory boot software application from the flash into the onchip RAM and then jumps to its entry point.
Factory boot application run

At this point the factory boot software application runs on the Nios II processor. It will query the flash objects looking for a valid application image to reconfigure the FPGA with. It locates the latest valid hardware application image and it elects to trigger a reconfiguration.
The Nios II programs the RSU controller with the appropriate address information and triggers the reconfiguration.
Reconfiguration

The reconfiguration controller now loads the application image into the FPGA.
Once the FPGA is configured with the application hardware image, the Nios II begins running from it’s reset location which happens to be the onchip ROM and RAM. This boot loader program attempts to locate an application software application in flash that can be loaded and executed in the onchip RAM.
Application load

The application image boot loader copies the application software application from the flash into the on-chip RAM and then jumps to its entry point.
At this point the application software application runs on the Nios II processor. It will start a console mode that allows a user to interact with it via the jtag uart, and it also listens for image downloads over the download jtag uart interface.
End of demonstration

At this point we’ve walked thru the process of programming the factory images into the flash and then downloading all the various combinations of software and hardware update images to illustrate how this particular remote update scheme would actually operate. While this scheme is quite robust and it offers excellent resistance to in field update failures, it is not intended to represent the only way that one can implement a remote system update scheme. The flexibility of the basic RSU hardware allows for much simpler schemes as well as more complex schemes to be deployed in the hardware.

If you run thru the actual demo on the BeMicro SDK hardware, it closes out by describing a couple of ways to inject failures into the application configuration flow such that you can observe the failure state in the factory boot image.