7. The Onboard Marvell Processor .................................................................41
   I. The U-Boot Boot Loader ........................................................................... 41
   II. The Linux Kernel .................................................................................. 41
   III. The Root Filesystem ............................................................................ 42
   IV. The startup.sh Script ........................................................................... 42
   V. DiniCmos ............................................................................................... 42
   VI. Filesystem Partitions ............................................................................ 42
   VII. Developing on the onboard Marvell CPU ............................................. 43
   VIII. U-Boot and Kernel Modifications ....................................................... 44
   IX. Setting the Root Password .................................................................... 47
X. Installing Alternate Operating Systems .................................................... 47

8. Using Emu On The Marvell ........................................................................48
   I. Configuring from a USB Flash-Drive or SATA Hard Disk ....................... 48
   II. Fixing An Unreadable USB Flash Drive ............................................... 49
   III. Configuring from the onboard NAND Flash ....................................... 50
   IV. Configuring From A Network Share .................................................... 51
   V. Debugging Flash-Drive and User Partition Configuration ..................... 52

9. PCIEDIRECT ConfigFPGA Cable Connection (High Speed Host-to-FPGA Data Transfer) ..........53
   I. Setting up PCIEDIRECT .......................................................................... 53

10. Board Stacking .........................................................................................55
   I. Four components of a Board Stack .......................................................... 55
   II. DNNMB_CONNECT Stacking System ..................................................... 55
   III. DNBC Stacking System ....................................................................... 56
   IV. Physical Assembly ................................................................................ 57
   V. Accessing stacked boards from Emu ....................................................... 57
   VI. Directly addressing stacked boards over FPGABUS ............................... 57
   VII. Cabled Connections Report/Verification .............................................. 58

11. Upgrading Software and Firmware ..........................................................59
   I. Software Updates .................................................................................... 59
   II. Device Driver Updates .......................................................................... 59
   III. Firmware Updates ................................................................................ 59
   IV. Recovering From Firmware Issues ....................................................... 60
   V. Linux Kernel Updates ............................................................................ 63
   VI. U-Boot Boot-Loader Updates ............................................................... 65
   VII. Updating the Recovery Partition ......................................................... 67
   VIII. Loading a Custom ConfigFPGA Image .............................................. 67
   IX. Cloning Firmware From Existing Boards .......................................... 68
12. FAQs, Gotchas, and Known Issues ........................................................................................................... 70

I. Ethernet hostname registration in Windows ........................................................................................................... 70

II. Hotplugging .......................................................................................................................................................... 70

III. PCIe Connectivity and Board Reset ..................................................................................................................... 71

IV. Multiple Boards On One Host PC .................................................................................................................... 71

V. Multiple EMULIB Connections To The Same Board .......................................................................................... 71

VI. Ultrascale SPI_FALL_EDGE SelectMAP loading .............................................................................................. 71

VII. Ultrascale board stacking with slave boards powered down ............................................................................ 71
1. Introduction

Emu is both an end user application for interacting with Dini Group hardware as well as a development kit for extending Emu’s capabilities or writing custom applications. Emu is designed to interface with any Dini Group board that includes a Marvell MV78200 processor. Emu compiles into both a command-line menu-system program (CMD version) and a graphical interface program (GUI version), supporting all functionality in both versions. The CMD version additionally supports scripting and command-line interaction, allowing customized interaction with the board right out of the box. Emu supports Windows and Linux platforms, using the QT windowing package for cross-platform support of native GUI interfaces. QT is freely available from the internet and is required for the GUI versions of Emu. It is recommended to use the QT environment in Emu development, but the CMD version can be built without it, using other standard environments such as gcc and MSVC.

The first part of this manual describes the basics of using Emu out of the box. It describes connecting to your Dini Group hardware, configuring FPGA’s, setting up clocks, and transferring data between the host and the user FPGA’s. The rest of the manual describes how to build the Emu application from source, and how to get started on your own Emu customizations or entirely new custom applications.

![Figure 1: Emu screen shot, showing a DNV7F2A](image)
2. Using Emu

This chapter describes the basic use of Emu for tasks ranging from FPGA Configuration to Board Test.

I. Choosing A Binary

The first step to using Emu is finding the right executable, or building one if you are working in a Linux environment. The Emu release binaries are found in “Software/emu/App/out_release”. The binaries are named clearly to reflect their usage:

emu_[gui/cmd]_[win32/linux]_[dbg].exe

For example, to run the GUI release version on a windows platform, use “emu_gui_win32.exe”. The “release_notes.txt” file contains a list of changes made in each release. The binaries will match the latest release described in the “release_notes.txt” file.

The GUI version of Emu is the easiest to use for human interaction with Dini boards. It gives a visual representation of the board and lets you click on FPGA’s to configure them, and provides a convenient menu system for all of the board features.

The CMD (command-line) version of EMU presents a text-based menu system and can do everything the GUI version can do, but without the pretty graphical interface. With the -c option the CMD version runs in command interpreter mode, taking commands and parameters from stdin, processing them, and producing output on stdout. This allows for advanced scripting of complex operations, which in many cases will allow users to automate complex tasks without touching a line of Emu code!

The “emu_mv” binary is Emu compiled for the onboard Marvell processor. This binary is preloaded on the Marvell filesystem to facilitate USB-Flash drive configuration and any other desired board interaction directly from the Marvell shell. See the chapter “Using Emu On The Marvell” for more information.

For Linux users, it is impossible to distribute binaries compatible with the wide range of systems out there, so it will be necessary to compile from source. Skip ahead to the chapter “Building Emu From Source” at this point, and return here once a working binary has been produced.

II. Choosing An Interface

Dini Group boards support a variety of interfaces for connecting to a host PC. All compatible boards offer ETHERNET, USB, and PCIE connectivity, all with a bandwidth of about 30MB/s to the user FPGAs. Many products additionally support a high speed “PCIEDIRECT” interface for getting high bandwidth (>500MB/s) between the user FPGAs and the host PC. The diagram on the following page illustrates in detail the various data paths supported by Emu.

All of the interfaces support all of the functionality, so the choice between them comes down to convenience and performance. If the application requires greater than 30MB/s performance, then the PCIEDIRECT interface is the only choice. If 30MB/s is sufficient, then ETHERNET and USB are much more convenient to use because they are both “hot pluggable”; they don’t require that the host PC is rebooted each time the Dini Group board is power cycled or reset. ETHERNET may require some setup to get an IP address properly assigned to the board, but has the advantage that the Host PC can be anywhere on the network, in another building, or even another country!
*All speeds shown are actual real world achievable data rates. MB=1 million bytes.

**FPGABUS is implemented with NMB on Virtex-7 and earlier boards, and with TMB on newer boards.
III. Device Drivers

This section describes how to install the device drivers that are required for Emu to communicate with a Dini Group board over certain interfaces. Source code is included for all device drivers, though it is unusual that anyone would want to modify the provided drivers. On Windows, drivers are required for PCIe and USB connectivity. On Linux, only PCIe requires a custom driver, but USB also has specific requirements—see below for details. Ethernet requires only that standard network drivers have been installed for your network card. If your computer can access an office network or internet connection, then it is ready to access a Dini Group board over Ethernet.

Windows: PCIe

Install the board into the PCIe slot, or connect the PCIe cable to the cable adapter card. Be sure that all required power connectors are connected (see board user manual for more information). Boot up the machine and let plug-and-play detect the board and prompt for a driver. Specify the driver location as:

```
[Customer Support Package]\Host_Software\[version]\Software\emu\Drivers\windows_pci
```

Allow the driver installation to complete. The board is now ready for use with Emu.

**Note:** If this interface uses an iPass cable connection on your board, then the host PC must be powered before the Dini board is powered on, to satisfy the Marvell CPU requirement that the PCIe ref clock is present at power on. Once the Dini board is powered on, then the host PC must be rebooted so that it will correctly enumerate the Marvell CPU at the start of the boot sequence.

Windows: PCIEDIRECT

Typically a PCIe cable card is required for this interface, consult the board’s user manual or contact support@dinigroup.com for more information. The board must be powered on and allowed to boot BEFORE the host PC is turned on, because the board is not able to be enumerated until the configFPGA is configured. Once the board is up, power on the host PC and let plug-and-play detect the board and prompt for a driver. Specify the driver location as:

```
[Customer Support Package]\Host_Software\[version]\Software\AETest\wdmdrv\drv
```

Allow the driver installation to complete. The board is now ready for use with Emu.

See the chapter dedicated to the PCIEDIRECT connection for more details.

Windows: USB

Boot up the windows machine without the board connected. Power on the Dini Group board and allow the processor to boot (wait about 1 minute). Connect the USB cable to the windows machine. Windows plug-and-play will detect the device and prompt for a driver. Specify the driver location as:

```
[Customer Support Package]\Host_Software\[version]\Software\emu\Drivers\windows_usb
```

Allow the driver installation to complete. The board is now ready for use with Emu.

Linux: PCIe

Install the board into the PCIe slot, or connect the PCIe cable to the cable adapter card. Be sure that all required power connectors are connected (see board user manual for more information). Boot up the machine. Find the linux driver and source at:

```
[Customer Support Package]/Host_Software/[version]/Software/emu/Drivers/linux_pci
```

There is a “README.txt” file there that describes how to build the driver for your linux distribution, and how to load and unload the driver once it is built. It is up to you to determine the appropriate place to source “dndev_load.sh” if you want your system to load the device driver on system startup.

**Note:** If this interface uses an iPass cable connection on your board, then the host PC must be powered
before the Dini board is powered on, to satisfy the Marvell CPU requirement that the PCIe ref clock is present at power on. Once the Dini board is powered on, then the host PC must be rebooted so that it will correctly enumerate the Marvell CPU at the start of the boot sequence.

**Linux: PCIEDIRECT**

Typically a PCIe cable card is required for this interface, consult the board’s user manual or contact support@dinigroup.com for more information. The board must be powered on and allowed to boot BEFORE the host PC is turned on, because the board is not able to be enumerated until the configFPGA is configured. Once the board is up, power on the host PC and allow it to boot. Find the Linux driver source at:

([Customer Support Package]/Host_Software/[version]/Software/AETest/linuxdrv-2.6)

There is a “README.txt” file there that describes how to build the driver for your Linux distribution, and how to load and unload the driver once it is built. It is up to you to determine the appropriate place to source “dndev_load.sh” if you would like your system to automatically load the device driver on system startup.

See the chapter dedicated to the PCIEDIRECT connection for more details.

**Linux: USB**

In Linux, Emu uses two mechanisms to attempt to connect to devices over USB. The first uses the “/proc/bus/usb” mount point of the “usbfs” (previously usbdevfs) file system driver. Until recently this was supported by most kernels. On these systems, the USB filesystem is mounted at “/proc/bus/usb”, typically by adding a line to a startup configuration file, such as after the “proc” entry in “/etc/fstab”:

    none /proc/bus/usb usbfs defaults 0 0

or by adding a line to “/etc/rc.sysinit”:

    mount -n -t usbfs /proc/bus/usb /proc/bus/usb

This last line can also be used from the command line to create the mount point after booting up.

To access the USB, Emu will have to be run as root unless the above mount command is altered so that usbfs is mounted with user permissions:

    mount -n -t usbfs -o devmode=0666 /proc/bus/usb /proc/bus/usb

Emu expects the usb filesystem to be mounted at “/proc/bus/usb”, and it is hard coded in “emulib_os_dep_linux.cpp”. If usbfs is supported but the mount point is not standard, then edit this source file before compiling Emu for your system.

Recently, kernel distributions have stopped including /proc/bus/usb mounted usbfs support in their default configurations. In this case, Emu tries a second method, making a system call to the “usb-devices” program to query for usb devices, and opening devices for access under “/dev/bus/usb/”. This method is compatible with the latest releases of most kernels.

Most systems will work without any modification to Emu or the kernel installation; simply connect the USB cable, open Emu, and select the board. If Emu cannot query the USB using either of the described methods it will display an error on startup. If you intend never to use USB and this error message is annoying, then edit the emu.ini file and set “discover_usb” to false, to prevent Emu from attempting to scan the USB bus.

Unless our USB device is specifically set up for regular user access, Emu will need to be run with root privileges in order to access the board. A script can be found in emu/Drivers/linux_usb that can be used with most Linux distributions to register our USB Device for regular user access.
IV. Finding Hardware
When Emu starts it will attempt to reconnect to the last known board [This behavior can be suppressed with the –m option, or by editing the emu.ini file]. If the board was not found, or there is no last known board, it will open with no board selected. Choose “Board->Select Board” from the menus and Emu will scan Ethernet, PCIe, and USB for supported Dini Group hardware. You will then be prompted to select a board from the list of detected boards. Choose your board based on its type, serial number, and interface. Once a board is selected, all of Emu’s features become available to set clocks, configure FPGA’s, and transfer data.

Ethernet

**KEY POINTS:**
- The default configuration uses DHCP to acquire network settings
- Static IP is supported but must be manually configured before use
- Ports 4950 and 4951 must allow UDP/TCP traffic in order to find boards
- Port 3490 must allow TCP traffic in order to talk to boards
- No special device drivers or root privileges are required
- Gigabit Ethernet is fully supported and recommended (1000BaseT)
- Avoid disconnecting or powering down devices while Emu is connected to them.

The Emu “Select Board” option uses a UDP broadcast command to detect boards on the local network. Emu detects each network adapter in your system and does a subnet broadcast on each adapter on port 4950. A subnet broadcast uses the broadcast address (255.255.255.255) qualified by the subnet address. So if your adapter has an IP address of 192.168.1.4 and a network mask of 255.255.0.0, then the subnet broadcast address is 192.168.255.255. Any Dini boards present on this subnet will respond to this broadcast message. Responses come back from the boards as TCP messages on port 4951 and are sent directly to the network adapter’s IP address. The “Select by IP” option does the same thing as the “Select Board” option but sends the UDP detection request directly to the specified IP address.

Make sure your host machine’s firewall allows UDP packets out on 4950 and TCP packets in on port 4951 for each physical network adapter on which Dini boards may reside, so that Ethernet board discovery can work.

Once a board is selected, regular board communication takes place on port 3490 using a TCP connection. **TCP traffic must be allowed in both directions on port 3490 for each physical network adapter on which Dini boards may reside, so that Ethernet board communication can work.**

In some network environments Emu may be unable to find connected boards with its Broadcast command, for instance if the Dini board is not in the local network and resides outside of a gateway router that does not forward UDP broadcasts. In this case it will be necessary to select the board directly by its IP address. Use the “Board->Select by IP” menu option.

By default the board acquires its IP address from the local network using DHCP. To determine the board’s IP address, connect to its “Marvell CPU serial port” (often labeled ‘RS232 CPU’) with a terminal program (19200bps, no parity, no flow control) and at the Linux prompt type ‘ifconfig’ to discover what IP address your board was assigned (or use the Board->System Call menu option in Emu and issue the ‘ifconfig’ command). If your network does not support DHCP then you will need to set your board up for Static IP. For current-session only setup ‘ifconfig’ can be used to configure the network settings. To
make the static IP settings persist, the board settings must be changed. Use a USB cable to connect to the board, or run “emu_mv” from the Marvell CPU terminal in order to change the board settings. Use the “Board->Set Board Info” option in Emu. Be careful not to change the board type or any of the FPGA stuffing information in these dialogs. Click “OK” in each dialog box until you come to the dialog box that contains the network settings. Uncheck the box for “Use DHCP”, and then set the Static IP, network mask, gateway IP, and DNS as appropriate for your network. Uncheck the box for setting the realtime clock, if it is checked. Accept all other defaults. Emu will disconnect from the board and the board will automatically reboot with the new settings. Once the board is booted you can check that it was configured with the new IP Address by using ‘ifconfig’ at the Linux prompt as described above. Then use either the ‘Board->Select Board’, or ‘Board->Select by IP” menu option to connect to the board in Emu.

Ethernet is “hotpluggable” meaning that you can connect or disconnect the Ethernet cable at any time. If the board is booted with the Ethernet cable connected (recommended), then it will bring up the interface right away and acquire a network address with DHCP if enabled. If the cable is connected after bootup it may take a few seconds before the interface is ready. The Marvell CPU serial port may be monitored to see what is happening or to debug connectivity issues. If the Ethernet cable is unplugged while Emu is connected to the board an API error will result and the software may become unstable. It is recommended to disconnect from the board or close the Emu software before disconnecting the Ethernet cable or powering down the board.

**USB**

**KEY POINTS:**
- In Windows the device driver must be installed
- In Linux, the USB device must be “registered” to allow regular users access to it, otherwise Emu must be run with root privileges to access the USB device
- High Speed USB 2.0 is fully supported and highly recommended
- Avoid disconnecting or powering down devices while Emu is connected to them

See the section above titled “Device Drivers” for information about installing the windows device driver. Linux does not require a device driver for USB, however most Linux systems will require that Emu is run with root privileges in order to access the USB bus, unless the device is set up for regular user access. See the “Device Drivers” section above for more details. Once the access is set up, simply plug in the provided USB cable, and choose “Select Board” from Emu. Like Ethernet, the USB interface is “hotpluggable”; the cable can be connected or disconnected while the board is powered on. If the cable is connected before the board is powered on, then the USB interface will be available at the time DiniCmos finishes initializing.

Also like Ethernet, it is highly recommended to disconnect from the board in Emu or to close the software before unplugging the USB cable or powering down the board. Otherwise a USB API error will result and the software may become unstable.

**PCIE**

**KEY POINTS:**
- In both Windows and Linux a device driver must be installed
- In Linux Emu must be run with root privileges to access the PCIe bus
Devices support Gen1 PCIe x4 connections, throughput to FPGA’s limited to ~30MB/s
Disconnecting or powering down devices while the host is running is not permitted

See the section above titled “Device Drivers” for information about installing the PCIe drivers for Windows or Linux. Emu will not be able to detect boards on PCIe until the drivers are properly loaded. PCIe is not “hotpluggable”, meaning that the board must be powered on at the time the host machine enumerates the PCIe bus and cannot be reset or power cycled without rebooting the host machine. If a cable connection is being used, the cable should not be disconnected while the board or the host machine is powered on. If Emu is not detecting your board, check to see if the board has been enumerated by the host (using “lspci” in Linux, or “Device Manager” on Windows). If the board was not enumerated then reboot the machine and be sure the board is powered on while the host is booting up. Because PCIe is not “hotpluggable”, it is recommended to use Ethernet or USB during project development to avoid frequently having to reboot the host machine. Because the EMULIB API does not specify what interface is being used, the interface can be switched at any time without changing a single line of code.

**PCIEDIRECT**

**KEY POINTS:**
- In both Windows and Linux the AETest device driver must be installed
- In Linux Emu must be run with root privileges to access the PCIe bus
- Devices support Gen2 PCIe x4 connections at roughly 600MB/s throughput
- A Dini Group iPASS cable card is used to connect the board to the host PC
- Not all Dini Group products have a PCIEDIRECT connection

See the section above titled “Device Drivers” for information about installing the AETest PCIe device drivers for Windows or Linux. Emu will not be able to detect boards on PCIe until the drivers are properly loaded. The PCIEDIRECT interface uses our AETest code base to get the highest possible PCIe bandwidth. Data to/from the user FPGAs bypasses the Marvell CPU bottleneck yielding true PCIe performance, while board management data is still routed to the Marvell CPU to preserve all of the Emu functionality. See the tips in the “PCIE” section above if trouble is encountered.

**Multiple Boards**

Emu is designed to work with one board at a time. If control of multiple boards is required, simply run multiple instances of Emu and select a different board with each. There is no hardware restriction to multiple board control within an application, so custom applications using the supplied EMULIB library code can be designed if this approach is desired.

**V. The Menu System**

![Figure 2: The Emu menu system as seen in the GUI build.](image)
The menu system in Emu puts all of the common operations right at your fingertips.

The “File” menu (GUI only) gives you an “Exit” option and a “Kill Process” option. Every menu selection is run as a separate thread. In the case that something goes wrong and the thread is not returning control to the main program, this option can be used to kill the offending process and bring the software back to a normal state. This option may be useful while developing custom Emu functions to aid in debugging or to abort a long operation.

The “Board” menu provides options for selecting and deselecting Boards as described above in the section on “Finding Hardware”. The “Display Board Info” option will show firmware revisions, fpga stuffing info, and other low level board data. When contacting Dini Group support ([support@dinigroup.com](mailto:support@dinigroup.com)) about a board, include this text in the message. “Set Board Info” is primarily for factory use, to set up board parameters before first-time use. It may be used to restore functionality in the event that the NAND Flash on the board is compromised. There is also a “system call” option here, which lets you execute shell commands on the onboard Marvell processor and see the returned text. Try issuing the “date” command or the “Is” command to see how it works. Remember that any command that blocks for user input will essentially lock up the board because the system call will never return- so be careful with this feature! If you lock up the system in this way, either power cycle the board or hit the “sys_reset” button and wait for the Marvell to reboot, then select “Board->Reconnect” to reestablish your connection. There are also options here for checking and/or updating the firmware and software for your board.

The “FPGA” menu provides options for configuring and clearing FPGAs, as well as issuing logic resets to configured FPGAs and checking the configuration status of FPGAs.

The “Clocks/Temps” menu has options for displaying temperatures and reading and setting the boards configurable clock networks. Temperature sensors provide data on the user FPGAs, the configFPGA, and the Marvell processor. Every Dini Board has its own clocking topography, and Emu provides an interface for setting the clock muxes and the synthesizer frequencies all in one place. See the user manual for your specific Dini product for more information on the clocking topology of your board.

The “Data” menu is the gateway to the FPGABUS, which provides a simple and efficient data transfer mechanism between hosts and user designs. Virtex-7/Stratix-V and earlier boards used NMB (LVDS based I/O) as the FPGABUS, later boards use TMB (transceiver based I/O). No matter the underlying technology, the Emu software interface is identical. Dini Group provides the endpoint module for users to include in their FPGA designs. Full source code and documentation is provided, as well as reference designs. Browse the FPGABUS address space, transfer to/from binary files, or peek at specific addresses all from this menu. Other address spaces are also found in the “Data” menu, including the ConfigFPGA register space. The ConfigFPGA register space is primarily for diagnostic and debug purposes and will not be of much interest to most users.

The “Test” menu contains a variety of diagnostic tests that verify correct operation of the Dini Group hardware. See the section below on “Running Field Tests To Verify Board Operation” for more information on using these tests.

The “Custom” menu is provided entirely for customer use. Its functions are implemented in the “App/source/custom.cpp” file, and are well documented to get you started adding your own custom
functionality to the software. In just minutes you could have a custom menu option to set global clocks, configure FPGA's, reset FPGA designs, and transfer a block of data over the FPGABUS!

The “Settings” menu can be used to modify Emu settings that appear in the emu.ini settings file. The emu.ini file is written out every time the program exits, to the same directory where the Emu executable is located. Some settings are only read in when the program starts, so when modifying settings from the menu system it is best to close Emu and then start it again to make sure all settings changes take effect. Settings may also be changed by editing the emu.ini file directly, just remember to close Emu BEFORE making changes since Emu will write out the file with its current settings when the program closes. Save emu.ini after making changes and then start the Emu program again to read in the new settings. See the section below on the emu.ini file for more details.

The “Settings” menu can be used to modify Emu settings that appear in the emu.ini settings file. The emu.ini file is written out every time the program exits, to the same directory where the Emu executable is located. Some settings are only read in when the program starts, so when modifying settings from the menu system it is best to close Emu and then start it again to make sure all settings changes take effect. Settings may also be changed by editing the emu.ini file directly, just remember to close Emu BEFORE making changes since Emu will write out the file with its current settings when the program closes. Save emu.ini after making changes and then start the Emu program again to read in the new settings. See the section below on the emu.ini file for more details.

The “帮助” menu option (GUI only) contains the “about” option which will display the version and compile date of the application. It also provides a link to the online version of this document found on www.dinigroup.com.

VI. The “emu.ini” File

The first time Emu is run, it will create a file called “emu.ini”. Each time the program starts it looks for this file and loads in all of its settings. When the program exits, it writes this file out next to the Emu executable. If Emu was run from read-only media, such as a CD-ROM or write-protected disk, then the emu.ini file will not be written and no settings will be saved. The emu.ini file contains settings and user preferences that customize Emu for local use. The file is plain text and can be edited with any text editor. The format of the file is one setting per line, with a comment describing each setting. Settings can be modified by changing their value and saving the file. Note that Emu must be closed before modifying the emu.ini file, otherwise changes will be overwritten when Emu closes and writes out its current settings. Examples of useful settings in emu.ini include whether to automatically connect to the last known board at startup, which interfaces to scan for boards, whether or not to save one-shot test logs, and much more.

VII. Basic Operations: Set Clocks, Configure FPGAs

The basic steps to getting a design up and running include setting up the global clocks, configuring the FPGA’s, and optionally issuing a logic reset. There are several methods to accomplish these basic tasks in the Emu software which suit different purposes.

The simplest method is to use the GUI version of Emu. Simply click on each global clock on the GUI display, and set its Mux source and/or its synthesizer frequency. Then click on each FPGA and choose “configure”. When all FPGAs are configured with the target design, select “FPGA->Reset All FPGAs”.

The same operations can be done from the menu options, which is how they would be accomplished in the CMD version of Emu. Select “Clocks/Temps->Set Clock Mux” to set the clock sources and/or “Clocks/Temps->Set Clock” to set the synthesizer frequencies. Use the “FPGA->Configure FPGA From Host” option to configure each FPGA, and the “FPGA->Reset All FPGAs” option to issue a logic reset.

Most users will want to automate these basic tasks, since they will likely be done many times over the course of developing a design. There are two ways to accomplish this: using Emu in command-interpreter mode and writing a shell script, or by adding code to the “Custom” menu and rebuilding Emu from source to include your changes. Both methods have their advantages, and can even be combined:
If you add functionality to the Custom menu, then your code can be run from a script by issuing the appropriate command such as “custom_1” to run the Custom 1 menu option. The shell script method can be very useful to configure a board automatically at power-up. Name the script “dini.sh” and placing it in the root of a USB flash-drive. Plug the flash-drive into the Dini board and the script will be executed each time the board boots. For more details on USB flash-drive usage, see the section in this document titled “Using Emu on the Marvell”. For more details on writing Emu shell scripts, see the section titled “Scripting with Emu”. For more details on adding code to the “Custom” menu or otherwise modifying the Emu software, see the section titled “Expanding Emu: Modifying The Source”.

VIII. Running Field Tests To Verify Board Operation
The “Test” menu provides access to a test suite that can verify correct operation of the Dini product. Choose the “Field Test” option to automatically run the full test suite. “Factory Test” and “Daughtercard Test” options, which are hidden by default, run additional tests that require specialized hardware only available at the factory- these options are not intended to be run by customers in the field (the options can be unhidden by checking the “show factory tests” and “show dcard tests” boxes in the Windows and Menus settings). The “Selected Tests” option can be used to choose specific tests to run, and additionally provides some useful debugging options such as ‘don’t set clocks’ and ‘automated mode off’.

The first time tests are run it will prompt you for the folder where you keep your Dini Group bitfiles. This is asking for the root folder of the bitfiles, so in the customer support package it would appear as: E:\FPGA Reference Designs\Programming Files

The chosen path is stored in the emu.ini file for future use. If it needs to be changed later the software should prompt you, but if problems are encountered you can edit the emu.ini file directly, or edit the “refdesign_bitfile_path” under Settings->Paths. See the section on “Emu.ini” for more information.

While tests are running you can press a key at any time to pause testing and get a list of options. By default tests will pause on error, but this behavior can be changed here. You can skip to the next test or abort testing entirely from this menu. Note: In the GUI build, the text window must be selected before it can accept keyboard input, so click on the text window before trying to enter keystrokes.

IX. FPGABUS Memory Space and “MainRef” Design Examples
The FPGABUS is the out-of-the-box way to transfer data between a host PC and a user FPGA design. The available bandwidth on the FPGABUS varies based on the high-level API connection (USB, Ethernet, PCIe, etc) as well as the underlying hardware implementation (NMB or TMB). However the Emu software interface to the FPGABUS is uniform across all implementations on all supported boards.
Summary of board-level IO busses on Dini Group FPGA boards:

<table>
<thead>
<tr>
<th>YMB (Yes Main Bus, MainBus)</th>
<th>A Parallel bus connecting all FPGAs in a system together. On early Dini Group boards the MainBus was used as the host-to-userFPGA data path (FPGABUS), but with the introduction of Emu a higher bandwidth solution was needed. The MainBus, now called YMB, still exists on most boards but is intended for customer use as part of the FPGA-to-FPGA interconnect and not as a host-to-userFPGA data path.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGABUS</td>
<td>A data bus that moves data between a host PC and a userFPGA design on a Dini Group board. There are currently two board-level implementations of the FPGABUS (a Dini Group board will implement one of them):</td>
</tr>
<tr>
<td>NMB (New Main Bus, Not Main Bus)</td>
<td>LVDS I/O making point-to-point connections between configFPGA and user FPGAs. NMB is found on the Virtex-6, Virtex-7, and Stratix-5 board generations.</td>
</tr>
<tr>
<td>TMB (Transceiver Main Bus)</td>
<td>High-speed serial transceiver I/O making point-to-point connections between configFPGA and user FPGAs. TMB is found on the Virtex-Ultascale, Stratix-10 and newer board generations.</td>
</tr>
</tbody>
</table>

Virtex-6, Virtex-7, and Stratix-5 boards use “NMB” as the underlying bus architecture for the FPGABUS. NMB uses LVDS I/O in point-to-point links between the configFPGA and the user FPGAs and achieves maximum speeds of approximately 750MB/s. Virtex-Ultascale, Stratix-10 and newer boards use “TMB” as the underlying bus architecture for the FPGABUS. TMB uses high-speed serial transceivers in point-to-point links between the configFPGA and the user FPGAs, which results in higher performance and frees up more I/O pins on the user FPGAs for interconnect and expansion cards. TMB achieves maximum speeds of approximately 4GB/s. Note that moving 4GB/s of data through an FPGA is extremely resource intensive and is not practical for most designs. Therefore by default we provide a 64-bit bus at 100Mhz (~750MB/s) to the user FPGA which is much more manageable. If higher bandwidths are required, contact support@dinigroup.com. Also note that bandwidths above 30MB/s are only available when using the PCIEDIRECT connection to the board, as opposed to USB or Ethernet.

To begin exploring the FPGABUS, the supplied “MainRef” reference design bitfiles can be used. Simply load the appropriate bitfile into the target FPGA. The bitfiles are found in the customer support package in the following folder:
FPGA_Reference_Designs\Programming_Files\[board name]\user_fpga\MAINTEST\[fpga type]\Some Virtex-6 boards require that the “G2” clock be set to 200Mhz in order for the NMB Bus to function properly, and a user reset should always be issued after the designs are loaded. Use “Test->Selected Tests” and select the “FPGABUS Blockram Test” to have Emu automatically set clocks to appropriate values, load the MainTest bitfiles, and issue the user reset.

Once the design is loaded, the FPGABUS bus functions under the “Data” menu can be used. First try “Data->FPGABUS->Check FPGABUS Ready”. This will print out the FPGABUS status for each installed FPGA. The FPGABUS memory browser allows display of FPGABUS memory locations and makes it easy to write dwords or bytes and get immediate feedback. The FPGABUS Advanced option provides a flexible interface for reading and writing blocks of memory space and optionally checking the read data. The file transfer functions can be used to transfer larger amounts of data. The read/write functions are basic tools for quickly writing or reading a block of memory.
To use any of these functions requires an understanding of the FPGABUS addressing scheme, and the specific address map of the MainRef design. The FPGABUS uses 64-bit addresses. The upper 8 bits are decoded by the bus master in the configFPGA and represent the FPGA index (0=FPGA A, 1=FPGA B, 2=FPGA C, etc.). The remaining 56 bits are for the target FPGA to decode. The Dini Group reference designs by convention use the next 8 bits (55:48) as a function select decoded as follows:

```c
// FPGABUS SELECTS
#define FPGABUS_SELECT_BLOCKRAM       (0x01)
#define FPGABUS_SELECT_SODIMM           (0x02)
#define FPGABUS_SELECT_INTERCON         (0x04)
#define FPGABUS_SELECT_REGISTERS        (0x08)
```

These definitions and definitions of some of the register locations available in the MainRef designs can be found in the Emu source file “EMULIB/diniboard.h”. For more detailed information about a specific board’s memory map reference the verilog source for that board’s MainRef reference design.

An example FPGABUS address could look like this:

```
0x01020000_00001000
```

This address targets “FPGA B” (upper 8 bits = 0x01), DRAM (next 8 bits = 0x02), at byte offset 0x1000.

To start exploring the FPGABUS space, try opening the FPGABUS Memory Browser, and type “j”, for “jump to address”. Emu prompts for the upper 8 bytes of address, and then the lower 8 bytes of address. Try jumping to the start of the blockram space and experiment with reading and writing data. If DRAM modules are installed in the SODIMM sockets, then the FPGABUS_SELECT_SODIMM can be used to access them as well.

It is worth mentioning that Emu will return an error message if an attempt is made to read or write to an FPGABUS address which does not exist. For instance, on a 2 FPGA board, any address with the upper 8 bits set to 0x2 or higher is invalid. The typical error in response to this is an FPGABUS timeout, so if this is seen double check the address used to be sure it was correct.
3. Building Emu From Source

The following build types are supported:

**Full Emu Application**

- Dynamically Linked Debug Build, Dynamically linked Release Build, or Statically Linked Release Build
- With or without AETEST_LIB linked in [provides PCIEDIRECT and DNReadBacker support]
- Windows or Linux Operating System (QT, Visual Studio, or gcc build environments)
- GUI or CMD front end [CMD is command line and can also be used as a scripting tool]

**Standalone EMULIB Library** [For linking to a custom application, details in “EMULIB Library” chapter]

- Board tests are not included [INCLUDE_BOARD_TESTS cannot be specified]
- With or without AETEST_LIB linked in [provides PCIEDIRECT and DNReadBacker support]
- Windows or Linux Operating System (QT, Visual Studio, or gcc build environments)
- Static or Shared Library (Shared Lib will have dependency on compiler libs, and possibly AETEST_LIB)

I. Finding the Source Code

The full source tree for the current Emu release is provided in a zip archive inside the emu.zip package (http://www.dinigroup.com/files/web Packs/emu.zip) or in the “Host Software” folder of the board support package for your Dini Group board. The archive is named for the Emu release number (such as “1.7.3.zip” for Emu version 1.7.3). Always use the latest Emu release as your starting point unless you have a good reason for using an older release. Inside the source tree archive you will find a reproduction of the Software tree as it appears in the Dini Group revision control system. To find the actual Emu source code traverse the tree:

X.X.X/Software/emu/App/source

The QT project file for the full Emu application can be found here:

X.X.X/Software/emu/App/project_qt/emu.pro

The project files for building the EMULIB library standalone can be found here:

X.X.X/Software/emu/App/project_emulib

II. The QT Environment

The QT development framework has been chosen for this project because of its robust cross-platform support, user friendly IDE environment, and its open source licensing. Anybody can download and install QT in windows or linux, and be developing in the Emu codebase in little time and with little effort.

We currently use QT5. QT4 still works, but QMAKE will output a warning about an unknown QT flag “QT_WIDGETS”, which can be ignored. To get QT5, go to http://qt.io and click on “DOWNLOADS”. Choose the “COMMUNITY” download (LGPL license) unless you plan to redistribute your custom application as a closed-source product (in which case you will need to purchase a QT license). For Windows, install the MinGW version which uses the MinGW toolchain for compilation. The VS (Visual Studio) version can be used but will require some modifications to the existing source code to account
for Microsoft’s quirky implementations of various standards (See the section below “Building in Visual Studio”). The MinGW version includes the MinGW compiler, so nothing extra is required, whereas the Visual Studio version requires that you have Microsoft Visual Studio installed.

The following sections describe how to set up QT for Emu development, and will help to jumpstart development on new projects. Note that QT is required to build the GUI version of the application, but the command-line version of the program can be built with g++ (or MinGW-g++ or MSVC on Windows) without any QT dependencies. Building in other environments is covered later in this chapter.

NOTE: AVG virus protection may complain about an xml file virus during QT installation; this is a known false positive and can be ignored according to users on the AVG forums.

III. The Config File (Switching between GUI and CMD builds)
The Emu config file, located in “emu/App/project_qt/config.pri”, controls the type of build that will be done. To switch between building the GUI and CMD apps, simply change the first line in the “emu/App/project_qt/config.pri” file. If set to EMU_GUI it builds the gui version, if set to EMU_CMD it builds the command-line version. After making any changes to the config.pri file, or to any .pro file, do a ‘Clean All’ from the build menu in QT Creator, and then “Run qmake” to reprocess the .pro files. If you forget to do this you’ll get a bunch of weird errors when you try to compile. This is also a good time to check that your run configuration is set up to run the executable you’re about to build (see ‘Run Configurations’ section).

The second setting in the config.pri file (INCLUDE_AETESTLIB) controls whether support for the PCIEDIRECT interface and FPGA READBACK feature are to be included (See previous chapters for a discussion on PCIEDIRECT). These features require the use of the AeTest library, which is our code base for high performance PCI-express communication. If INCLUDE_AETESTLIB is specified, then the project will also build the AeTest library from source using the QT project in “emu/App/project_aetest_library/aetest_library.pro”. For users that do not wish to use these features, commenting out this line will skip building and linking to the AeTest library, simplifying and speeding up the build process and resulting in a smaller binary executable.

The last setting in the config.pri file is INCLUDE_BOARD_TESTS. For normal Emu application builds this should be used so that all of the board diagnostic tests are included. However, when building EMULIB for linking with external programs this should be commented out. The board diagnostic tests have many dependencies in the Emu application code-base, and so prevent the EMULIB library from being usable in external programs that don’t define all of these dependencies. This also prevents EMULIB from being built as a shared library, since shared libraries can’t have undefined symbols in them. Commenting out INCLUDE_BOARD_TESTS removes these dependencies and allows EMULIB to be used successfully as a standalone library. Note that the full Emu application can be built without INCLUDE_BOARD_TESTS if desired, in which case the “Test” menu will not be present and none of the code in the “TESTFUNCS” folder will be used. The produces a smaller Emu application binary and may be desirable if extending the Emu application for custom use, where the board diagnostic tests are not required.

IV. The Project Files
There are several different project files that may be used for building different components and from different platforms.
The main project file for building Emu with QT Creator is located in “emu/App/project_qt/emu.pro”. This project file simply calls out the dependent project files for building the AETest library (if PCIEDIRECT support is included), the EMULIB library, and the Emu application itself. The project file for the Emu application is in the same folder, in the file emu_app.pro. This file specifies all of the source files for each type of build (ie linux, windows, command-line, gui). It specifies where the binaries will go and what they will be called, and it specifies the external libraries that will be linked in. The config.pri settings file is also in this folder, which is discussed in the previous section.

In the folder “emu/App/project_emulib/” are projects for building the EMULIB stand-alone library from Qt Creator (emulib.pro), using gcc (Makefile.gcc), or using Visual Studio’s nmake tool (Makefile.nmake). There is also a small sample application here that shows how to link to the Emu Library and do basic board operations using the library.

In the folder “App/project_mv” is a Makefile for building Emu on the Marvell MV78200 processor. The board comes with the “emu_mv” binary pre-loaded on the Marvell filesystem. See the chapters “The Onboard Marvell Processor” and “Using Emu On The Marvell” for more details.

Also in the “App/project_mv” folder is a “Makefile_linux” which may be used to build the command-line version of Emu on desktop linux distributions without any QT dependencies.

V. Build Types: Debug vs. Release, Static vs. Dynamic

First off, let’s understand what all of these mean. A “Debug” build includes debug symbols in the binary code which allows us to step through code with a debugger, at the cost of producing a much larger binary. A “Release” build does not include debugging symbols and is built with compiler optimizations turned on to produce a smaller and faster binary. “Dynamic” and “Static” refers to how the binary is linked to the libraries it depends on. “Dynamic” linking uses shared libraries rather than including the library code inside the binary itself. In this case we are talking about the QT libraries, the MinGW C++ and runtime libraries, and the Dini Group provided AETest Library. Note that we always link statically to the EMULIB library due to dependencies to the board test functions in the Emu application code, and because there is no real benefit to dynamically linking this library. “Dynamic” linking produces a smaller binary in less time, but the resulting binary depends on external library files that must be present on the target machine for the binary to run (On Windows these are *.dll or “dynamic linked library” files, in Linux these are *.so or “shared object” files). “Static” linking includes the required library code in the produced binary, generating a much larger binary, but a binary that can run on any target machine without depending on external library files. Note that even our “Static” builds will still depend on standard system libraries that are expected to be present everywhere, for example the Windows sockets library “ws2_32” would never be statically linked into an application.

QT Creator allows you to select between different build configurations using the build selector, which is located just above the green arrow on the bottom left-hand side of the screen. It can also be accessed from the menus by doing ‘Build->Open Build/Run target selector…’. The selector allows you to choose which installation of QT to use for the build, and whether it is a “release” build or a “debug” build. Generally speaking, all development should be done with “Debug” builds, and should be dynamically linked. Building “Release” binaries produces more optimized results, but limits your ability to use the debugger to track down bugs. [NOTE: READ THE SECTION ABOUT STATIC LINKING BEFORE BUILDING RELEASE BINARIES!]. Different QT SDK packages come with different versions of the QT and Runtime
libraries, and which versions are included has changed as QT has evolved. You may get debug libraries pre-built or you may get release libraries pre-built, or you may get both. The provided libraries are all for dynamic linking (See the section below about static linking for details on creating statically linked builds). If your debug build fails due to the libraries not being found (ie QtCored, QtGuid) then try a release build to see if those libraries were provided (ie QtCore, QtGui). You may need to edit the emu_app.pro file to remove --static from the release build command.

For release builds, we have set up emu_app.pro to include --static on the link command line. Note that you must build the static versions of the QT libraries or your application will still link dynamically to those libraries. If you want different behavior in your release builds, modify the emu_app.pro file to get your desired settings. Note again that the QT libraries must be built for static linking before a static build will succeed. See the section below on setting up QT for static linking. If you want to produce dynamically linked release builds, comment out the --static line in the emu.pro file. If you want to switch between static and dynamic linking, you will need two installs of QT, one built for static linking and one built for dynamic linking.

VI. Compiling
By default, QT will use g++ for compiling and linking, and on Windows will use the mingW toolchain, which is installed by default along with QT. In order to be able to compile from the QT-Creator IDE, you may need to add the path to the mingW compiler to your PATH environment variable. The path is something like “C:\Qt\tools\mingw\bin”. Other compilers may be used, but with limited support from Dini Group. The command-line build of Emu can be built quite easily with g++ without any dependencies on QT. Microsoft Visual Studio can also be used, but some source code changes are required to get the build to work. If building the GUI version in Microsoft Visual Studio, the QT libraries must be built in Visual Studio- be sure to install the Visual Studio versions when installing QT. See the section at the end of this chapter for “Building In Visual Studio”.

VII. Run Configurations
Because we are building at least 4 versions of the program (or 5 if we build for Marvell MV78200 CPU), we are not calling the output file emu.exe. Instead we have emu_win32_cmd_dbg.exe, and emu_linux_gui_dbg.exe, etc. These names and the “App/out_development” and “App/out_debug” folder locations that they reside in are specified in the project file “App/project_qt/emu_app.pro”.

In order to have QT Creator run the application after building it (ie by hitting the green arrow button) you may need to edit the default run configuration. This is done in the ‘Projects’ pane of QT Creator. Select the build configuration you are using and then select “Run” at the very top under “Desktop”.

Any time you switch between GUI and CMD or between Release and Debug you will need to set up the run configuration to run the appropriate executable. You will also need to run “Build->Clean All” and “Build->Run QMake” to complete the switch-over.

VIII. Dynamic Linking and Missing Libraries (*.DLL or *.so)
By default, QT applications dynamically link to the QT libraries. This provides faster builds and smaller binaries. When running the app from outside the QT-Creator IDE, the operating system may not find the required QT library files (*.DLL in Windows, *.so in Linux). This will also be an issue for the Dini Group AE Test library if linking dynamically. In Windows this can be fixed by finding the missing *.dll or *.so
files and copying them to the same folder as the Emu executable. In Linux you’ll need to copy them somewhere in your library path, like “/usr/lib”, or add a new path to the LD_LIBRARY_PATH environment variable. The aetest library will be found in AETest/aetest/laetest_library.dll (or .so). QT libraries will be found inside the QT installation, typically in QT/[date]/qt/bin. You will need QtCored.dll (or .so) and QtGuid.dll (or .so) and perhaps a few others. Depending on your build settings you may also need the MinGW C++ and runtime libraries which can also be found inside the QT installation.

IX. Plugins
QT supports “plugins”, which are kind of like shared libraries. The “plugin” folder path is hard-coded in the QT core library. QT recommends calling QApplication::addLibraryPaths() in your program to add more search paths to find the plugins that you use. Emu does not use plugins because static linking cannot be done for programs that use them. Note that by default JPEG support is a plugin, and if the plugin is not found images simply aren’t displayed! For this reason all images used in the program are in PNG format, which is included in the QT library by default.

X. Static Linking
We statically link the QT libraries and compiler libraries for the Release builds of Emu, so we can distribute a single executable without worrying about shared libraries (DLL’s) and plugins. [note: plugins cannot be used with statically linked programs] In order to statically link the QT libraries, QT must be recompiled on the target platform. There is a wealth of information on this topic in this article: http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler/

Note that in this case when we talk about “static linking” we are mainly talking about linking to the QT libraries and gcc/MinGW libraries statically. The binary we produce will still have shared library dependencies to standard components, such as the MSVCRT.DLL on Windows. These components can reasonably be expected to exist on every platform, so are not of concern to us. To see a complete list of shared library dependencies run “Dependency Walker” on the produced Emu binary.

Notes on building statically linked Qt 5.12 libraries on Windows
Following are the steps that I followed to get static linking set up on Windows 7 with QT 5.4.1. The list of dependencies reported by dependency walker when I was done looked like this:
[Note: These DLL’s depend on many other libraries- this list shows the direct dependencies only]

Reported by Dependency Walker on "emu_gui_win32.exe":
c:\windows\system32\ADVAPI32.DLL
c:\windows\system32\GDI32.DLL
c:\windows\system32\IMM32.DLL
c:\windows\system32\KERNEL32.DLL
c:\windows\system32\MSVCR100.DLL
c:\windows\system32\OLE32.DLL
c:\windows\system32\OLEAUT32.DLL
c:\windows\system32\SETUPAPI.DLL
c:\windows\system32\SHLL32.DLL
c:\windows\system32\USER32.DLL
c:\windows\system32\WINMM32.DLL
c:\windows\system32\WS2_32.DLL
Reported by Dependency Walker on "emu_cmd_win32.exe":
c:\windows\system32\KERNEL32.DLL
c:\windows\system32\MSVCR100.DLL
c:\windows\system32\SETUPAPI.DLL
c:\windows\system32\USER32.DLL
c:\windows\system32\WS2_32.DLL

First, download the online installer. (http://qt.io download). At the time of this writing the latest version is “Qt 5.4.1 for Windows 32-bit (MinGW 4.9.1, 856 MB)“. Run the installer and check boxes for “MinGW 4.9.1“ both under “Qt 5.4.1” and under “Tools”. The “Qt Creator” box is forced to be checked. This will install Qt Creator along with the MinGW compiler and pre-built dynamically linked QT libraries. At this point you are ready to go for dynamically linked debug builds. You can also do release builds from this install that are dynamically linked to the QT release libraries.

Next, run the installer again, but this time select the “Source Components” for QT, and uncheck the “MinGW 4.9.1“ boxes both under “Qt 5.4.1” and under “Tools“. Choose a different folder to install into than used for the previous install. For me, the original install went into “C:\Qt5”, and I put the second install into “C:\Qt5_static”, to denote that the installation will be configured for producing statically linked release builds only. Note you will continue to run QT Creator from the original installation, and this new installation will be used only for building the statically linked QT libraries from source.

Make sure the mingw32/bin folder is at the BEGINNING of your path environment variable. My build failed bizarrely because my mksnt folder came first in my path and its version of ar.exe did not support the command-line options used by mingw-make while building. For me, the bin folder was located at “C:\Qt5\Tools\mingw49_32\bin” and I put this at the front of my PATH variable. Even with this in place, I still had to move my “unix_tools” folder out of my path because the QT Makefiles were using tools in there that had slightly different flags or behaviors than what it was expecting- so be careful!

Open a command prompt and go to c:\QT5_static\[version]\Src\qtbase

> mingw32-make confclean
If you have not done a configure on this QT installation yet, then this is not necessary, and will fail with a message like “no rule to make target ‘confclean’”. Otherwise, wait for it to delete files.
When it’s done do:
> configure -release -static -opensource -nomake examples -nomake tools -force-asserts -no-sql-sqlite -no-libjpeg -no-opengl -no-gif -no-angle

Windows only: -no-incredibuild-xge -no-dbus -strip -no-openssl -no-audio-backend -no-qml-debug

Note that in minGW, exceptions require dynamic linking to the MINGW10.DLL non-standard library. For this reason, we do not use exceptions in Emu, and use assert() instead to halt program execution when something is catastrophically broken. This is the reason for the -force-asserts flag.

Accept the licensing terms. We are using the LGPL opensource license, as specified on the command line above. Wait for this to complete [took 2 minutes for me]. Finally, do:
> mingw32-make sub-src
Now wait for a long while for this to complete [took about 30 minutes for me on a fast machine].

In order to build Emu with the newly built static release QT libraries, QT Creator must be set up to use them. If you run the QT Creator that was installed with this QT install, then it should use the correct QT install by default. Otherwise you may need to point QT Creator at this QT library install:

Tools->Options->Build & Run->Qt Versions, click Add...

Select the qmake.exe from the new version, like: “C:\Qt5_static\5.4.1\Src\qtbase\bin\qmake.exe”

Go to the “Kits” tab and select the new QT version, which should now appear under “Manual”

Assign the MinGW compiler to it if it didn’t automatically pick it up (“C:\Qt5\Tools\mingw49_32\bin”).

Give the kit a descriptive name so you can tell it apart from the dynamic debug version. Set the “Qt version” to use the version we just built. For me it was listed here as “Qt 5.4.1 (Src)”. Click “OK” to apply all these changes.

Finally, select the “Projects” tab on the left and do “Add Kit”, and select the new kit you just set up.

QT Creator is now set up for both dynamic debug builds and static release builds. To switch between them use the build selector on the bottom left of QT Creator just above the green arrow “run” button.

Click on the build selector to choose the kit and build type (ie release/debug). When switching between kits and/or build types ALWAYS do a “Build->Clean All” and “Build->Run QMake” to be sure everything is built correctly.

Note that the compiler libraries are still linked dynamically by default, which would require us to distribute dll’s like MINGWM10.DLL and LIBCC_S_DW2-1.DLL. We add -static to the linker command line for release builds to link to these libraries statically. This is done by adding this line to the .pro file:

CONFIG(release,debug | release):QMAKE_LFLAGS += -static

**Notes on building statically linked Qt 5.12 libraries on 64-bit Linux**

These notes are from 64-bit Linux Mint Cinnamon 19.1. Other distributions will require similar steps but some commands and library names may differ. The QT documentation website (docs.qt.io) has a lot of information about building QT from source and should be studied before reviewing the notes below.

Just as for windows, the first step is to download either the QT source package or the QT Installer and check the box for source code when doing the install. The second step is to install all of the library packages that will be required for building QT. Below is the apt install command used for a fresh Linux Mint installation to get everything needed to build QT. Some of these will already be installed and up-to-date, but including them all makes sure:

```
sudo apt install g++ clang++ build-essential llvm libgl1-mesa-dev libfontconfig-dev libx11-dev libx11-keysyms1-dev libxext-dev libxfixes-dev libxi-dev libvpx-dev libxrender-dev libx11-xcb-dev libxcb* libxkb*
```

Next we need to configure QT with the options we need. From a linux prompt, cd into the source code directory, something like ‘Qt5.12.1/5.12.1/Src’. The “configure” tool will be here. You can run ‘./configure -h’ to get a full list of the possible configuration options. Below are the options we used to build in all of the features required by Emu:
but since these libraries are pretty standard and are explicitly built static Qt version built statically along with the rest of dependencies means that in order to static link any Qt application, the application must be built statically along with the rest of the Qt libraries. As of this writing, the Emu project files have been updated to handle this. Another thing is that some libraries above are still using the shared versions, such as libpng16.so and libz.so.1, even though I explicitly built static Qt versions of these libraries. With some effort these extra dependencies could probably be eliminated, but since these libraries are pretty standard and are pre-installed on most Linux distributions it probably isn’t worth the effort.

After building the gui release version of Emu, run “ldd” on the resulting binary to see the library dependencies, and note the absence of any Qt specific libraries. It should look something like the following:

```
linux-vdso.so.1 (0x00007ffda9d9000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fd3181a4000)
libfontconfig.so.1 => /usr/lib/x86_64-linux-gnu/libfontconfig.so.1 (0x00007fd317f5f000)
libfreetype.so.6 => /usr/lib/x86_64-linux-gnu/libfreetype.so.6 (0x00007fd317cab000)
libX11-xcb.so.1 => /usr/lib/x86_64-linux-gnu/libX11-xcb.so.1 (0x00007fd317aa9000)
libxcb.so.1 => /usr/lib/x86_64-linux-gnu/libxcb.so.1 (0x00007fd317881000)
libXrender.so.1 => /usr/lib/x86_64-linux-gnu/libXrender.so.1 (0x00007fd316f99000)
l glimpse.so.0 => /lib/x86_64-linux-gnu/libglimpse.so.0 (0x00007fd316f5a000)
libX11.so.6 => /usr/lib/x86_64-linux-gnu/libX11.so.6 (0x00007fd31733f000)
libGL.so.1 => /usr/lib/x86_64-linux-gnu/libGL.so.1 (0x00007fd316f99000)
libGL.so.1 => /lib/x86_64-linux-gnu/libGL.so.1 (0x00007fd316f99000)
l bm.so.6 => /lib/x86_64-linux-gnu/libbm.so.6 (0x00007fd316fa1000)
l bxml.so.2 => /lib/x86_64-linux-gnu/lbxml.so.2 (0x00007fd316f99000)
l ibxmlReader.so.0 => /lib/x86_64-linux-gnu/lbxmlReader.so.0 (0x00007fd316f99000)
l ibxml.so.2 => /lib/x86_64-linux-gnu/libxml.so.2 (0x00007fd316f99000)
libxml-config.so.4 => /lib/x86_64-linux-gnu/libxml-config.so.4 (0x00007fd316f99000)
l ibxml2.so.3 => /lib/x86_64-linux-gnu/libxml2.so.3 (0x00007fd316f99000)
l ibxml-valid.so.4 => /lib/x86_64-linux-gnu/libxml-valid.so.4 (0x00007fd316f99000)
l ibxml-d1.so.3 => /lib/x86_64-linux-gnu/libxml-d1.so.3 (0x00007fd316f99000)
l ibxml-d1-validate.so.4 => /lib/x86_64-linux-gnu/libxml-d1-validate.so.4 (0x00007fd316f99000)
l ibxml-d1-catalog.so.5 => /lib/x86_64-linux-gnu/libxml-d1-catalog.so.5 (0x00007fd316f99000)
```

A few things worth noting. First is that at some point Qt started including xcb as a plugin rather than a shared library. This means that in order to statically link any Qt application, the application must explicitly import the xcb plugin, which must be built statically along with the rest of the Qt libraries. As of this writing, the Emu project files have been updated to handle this. Another thing is that some libraries above are still using the shared versions, such as libpng16.so and libz.so.1, even though I explicitly built static Qt versions of these libraries. With some effort these extra dependencies could probably be eliminated, but since these libraries are pretty standard and are pre-installed on most Linux distributions it probably isn’t worth the effort.
XI. What Went Wrong?
Use the “depends” tool (freely available on the internet) to verify that the executable is depending on the expected DLL’s and shared libraries. If the release version depends on anything in the QT folder then something is not right. If the debug executable is giant (>100MB) then you have probably linked statically to the debug libraries, which is not recommended! If mingwm10.dll is the only incorrect dependency it may be because you have used exceptions in your code, which prevents static linking of this library. It may be possible to overcome this, but that is beyond the scope of this document. Our solution is to not use exceptions in the application.

XII. Building in Visual Studio
We highly recommend using QT. It is free, easy to use, and everything is set up for you already. If you absolutely MUST use Visual Studio then this section may help you- but don’t expect it to be an easy out-of-the box solution: that is what QT is for.

A few source code modifications must be made due to differences in the standard library implementations of gcc and Microsoft Visual Studio. The problems that are encountered and their solutions are listed below:

**Problem:** inet_pton redefinition; different type modifiers
**Problem:** inet_ntop 2 overloads have similar conversions
**Solution:** In MinGW and some Visual Studio versions the inet_pton and inet_ntop functions are not defined, so Emu creates them. However, if your Visual Studio version does define them then you will see these conflicts. Comment out the inet_pton and inet_ntop declarations in emulib_os_dep.h and definitions in emulib_os_dep_win32.cpp. This will let Emu use the Windows versions of these functions.

**Problem:** open(), close(), read(), and write() are not found in diniapi_direct.cpp.
**Solution:** Add #include <io.h> to the top of diniapi_direct.cpp.

**Problem:** mkdir() does not take 1 parameter in emulib_os_dep_win32.cpp
**Solution:** Comment out the Emu mkdir() function- it is not needed for the Visual Studio build.

**Problem:** snprintf is not found in diniboard_id.cpp.
**Solution:** change all snprintf statements to _snprintf

**Problem:** stdint.h is not found.
**Solution:** This file was added in Visual Studio 2010. For older versions, download stdint.h from the internet and put it in an appropriate location.

**Problem:** interconnect_data.cpp: elements of partially initialized array must have a default constructor.
**Solution:** Delete this file, it uses partially initialized static arrays which are supported by gnu toolchains but not by the Microsoft tools. Without this file the “single intercon test” will not report pin and net information on errors, just the bank and index from the Verilog implementation.
**Problem:** `emulib_os_dep_linux.cpp` gets many errors and warnings.

**Solution:** Delete this file, it is for Linux builds only and should not be included in a Windows build.

Compiling the Command-Line version without QT is not too difficult, because no QT classes have been used outside of the GUI implementation. This means after making the syntax fixes above the program should build without any other extra steps.

Compiling the GUI version is a bit more involved, as the QT libraries must be installed and linked in to the Visual Studio project. Basic instructions for this follow.

1. Download and install the Qt5 for Visual Studios for your windows machine:

   Go to: [http://qt.io/download](http://qt.io/download)
   Download the COMMUNITY version of the online installer.
   Run the installer and select the Visual Studio version (Not the MinGW version)
   Include the Visual Studio Add-In if there is a check box for it
   Accept all other default values (If you change the install location, just make note of where it is)

2. Configure Qt to run with the Visual Studio compiler:

   Run the visual studio command prompt to set environment variables needed by the qt configure executable. This can be found from the start menu:

   start->All Programs->Microsoft Visual Studio->Visual Studio Tools->Visual Studio Command Prompt

   Now go to the path of your installed qt:

   e.g. “C:/Qt/5.4.1/qt”

   And type

   > configure

   When asked for commercial or open source, type ‘o’
   When asked if you read the documentation, type ‘y’
   Now wait while Qt configures itself to the Visual Studio Environment
   When completed type

   > nmake

   Wait an even longer time while Qt rebuilds itself for Visual Studio

3. Configure the Visual Studio Add-In if it was installed with QT:

   After the add-in is installed, open Visual Studio
   Go to the “QT” menu as follows: Qt > Qt Options
   Make sure that there is a Qt version and it matches the one you installed and configured
   If it does not match, click ‘add’ and put the correct version number and path
   e.g. version = 5.4.1, and path = C:/Qt/5.4.1

   Now you are ready to build the entire GUI application.

**XIII. 64-bit Builds**

**Linux**

In 64-bit Linux, all Emu builds will be 64-bit. Use the 64-bit Linux QT SDK on your 64-bit Linux OS and everything will just work. For PCIe, the linux PCIe driver needs to be built and loaded using the same steps as would be done in 32-bit Linux. There are no special steps required for 64-bit Linux.
**Windows**

The demand for 64-bit Windows apps has not yet grown enough for QT to provide an out-of-the-box 64-bit Windows distribution. The Windows version of the QT-SDK will use the MinGW compiler which only produces 32-bit binaries. Of course the SDK and the 32-bit binaries it produces will run just fine on any 64-bit (or 32-bit) Windows machine and for the vast majority of users there is no reason to worry about it. There are some instances, however, where a 64-bit build may be needed, for instance if the EMULIB API is to be used from an existing 64-bit application. To create a 64-bit version of the EMULIB library, simply follow the instructions for building the stand-alone EMULIB library using Visual Studio, but do the builds from the 64-bit windows machine using the 64-bit visual studio command-prompt. No other special steps are required. If Visual Studio is not available, it is probably possible to use Mingw-64 in place of Mingw to produce 64-bit binaries. Compiling the entire Emu application as a 64-bit binary can be accomplished pretty easily for the CMD build using Visual Studio as described above. Building the GUI version into a 64-bit binary is more difficult because it requires 64-bit versions of the QT libraries. The QT installer now has 64-bit Visual Studio libraries, so it is recommended to use Visual Studio for 64-bit builds. The other option is to rebuild the QT libraries from source using Mingw-64 which may be rather tricky and is way beyond our ability to support, but some help may be found through extensive use of Google.
4. Expanding Emu: Modifying The Source

Before making any modification to the source code, read the previous chapter on compiling Emu from source and be sure you can build and run a working binary from the existing code. Once you have accomplished this, then read through this section for help in understanding the basic structure of the application and for tips on how to get started customizing the Emu application. Some customers will prefer dropping the high level Emu code and using the EMULIB API library directly instead- see the chapter “EMULIB Library” for more information on this topic.

I. The “Custom” menu

The first step to modifying the emu source code is to make a small change to one of the custom menu options, rebuild the application, and see the change take effect. The file to edit is “App/source/custom.cpp”. This file contains basic examples of how to perform user I/O in the program and how to interact with the Dinigroup hardware. In the GUI build, the “Custom” menu is found in the top menu bar as a drop-down menu. Try selecting the options to see the results. In the CMD build, the option is displayed in the top level menu. In many cases, customers can get all the custom functionality they need simply by adding their code to an option in the custom menu. Other customers will want to create an entirely new application, keeping only the hardware interface part of the provided source code (called “EMULIB”); tips for this type of customization are found in the chapter titled “EMULIB Library”.

II. Basic Application Structure

The application is divided into five distinct units. At the bottom is “EMULIB”, contained in the “source/EMULIB” folder. This piece provides a low level API for interacting with the Dini Group hardware. See the chapter “EMULIB Library” for more about the library and how to use it. At the top are the user I/O units, contained in the “GUI” and “CMD” folders. These implement the user I/O API that is documented in the “emu.h” file in the main folder. The GUI build uses the “GUI” implementation and the command line build uses the “CMD” implementation. The “TESTFUNCS” folder contains a unit dedicated to hardware verification called “Oneshot Test”. This provides a suite of tests, some of which are run only at the factory and others that can be run in the field for customer verification. The “Test” menu in Emu contains all of the options implemented in this unit. Commenting out the “INCLUDE_BOARD_TESTS” line in the config.pri file removes this unit from the build. Finally, the main program unit, which is comprised of the files in the “source” folder itself. This unit builds the menu system (in “menu_system.cpp”) and provides the interface between the high level user I/O and the low level EMULIB API.

III. Emu I/O System

When working in the high level Emu code, “Emu.h” describes all of the available I/O functions for displaying text and for interacting with the Dini Group hardware. This is an abstraction layer above the EMULIB library that handles user I/O and error reporting. This abstraction allows the program to be compiled as a GUI or a Command-Line application without any changes to the source code at all.

Another useful tool is the ini_settings system, which generates and reads in the “emu.ini” file. This settings file always resides next to the Emu executable and contains settings for things like last-used
FPGA programming files, window size and location, and auto-board selecting options. See the top of “ini_settings.cpp” for the list of implemented settings. New settings can be added here as well to support new features.

IV. Adding One-Shot Tests To Emu

One of Emu’s primary functions is to provide a testing platform that allows both factory and field testing of Dini boards. The goal is to provide a system that is easy to use, flexible, and provides clear feedback as to the status of the various tests. In order to provide these features certain rules must be followed when adding tests to the One-Shot system. These rules are outlined in this section.

a. Every test has the same function prototype:
   
   ```cpp
   bool func_name(oneshot_state_t& state);
   ```
   
   The function returns true if it passed, or false if it failed. The function must not directly access any fields in the “state” parameter. This parameter is used to interact with the One-Shot support functions described below.

b. Declare the function in “TESTFUNCS/oneshot_tests.h” in the appropriate section: FACTORY TESTS or FIELD TESTS. The factory tests are tests that require special hardware that is not provided to the customer. Field tests are tests that customers can successfully run in the field with the standard equipment.

c. Add the test to the appropriate boards in the static “BOARD_INFO” structure in “EMULIB/diniboard.h”. Search for “factory_tests” or “field_tests” to find the appropriate column where the tests are listed. Add an entry for the new test using the “EMU_TESTFUNC” macro. The order the tests are listed here determines the order in which they will run, so choose intelligently where in the list it goes to minimize how many times the FPGA’s must be reconfigured with different designs (If the correct design is already loaded it will not reconfigure).

d. Add the definition of the new function to an appropriate .cpp file in the TESTFUNCS folder. Try to group similar tests into the same .cpp file so we can limit the number of files here to a manageable number. Only create a new .cpp file if your test really doesn’t fit in any of the existing categories, and then create a file that is general enough that future tests of similar nature can be included in the new file.

e. Follow the example of existing test functions!

f. Every test function must check that “global_selected_board” is not NULL before doing anything, and print “NO BOARD SELECTED” and return false immediately if it is.

g. Every test function must call “oneshot_is_automated(state)” to determine if the test is running in automated mode or manual mode. In automated mode the test must not pause for user interruption, and must do the most exhaustive and complete test possible. This is the mode used when we do our factory test, so it better fully test anything this function is supposed to cover. If in manual mode, then the user should be given options to select specific FPGA’s and other parameters to aid in debugging when things aren’t working right.

h. Every test must call “oneshot_prepare_test(state, folder, design_type, fpgas, clock_settings)” to guarantee the correct settings will be used for the test. If the test was run with the “don’t set clocks” option, then the oneshot_prepare_test() function will not change the clock settings and
will instead display the current clock settings. If this function returns false the test should report the error and return false.

i. Every test must report errors with the “oneshot_reporterror(state,message)” function. After reporting an error the test can either return false immediately, or check the return value of oneshot_reporterror() and return false if oneshot_reporterror() returned false, or continue testing if oneshot_reporterror() returned true. Only allow continued testing if it makes sense that someone may want to attempt continuing in that situation.

j. Every test must use oneshot_fpgabus_read() and oneshot_fpgabus_write() when accessing the board. This function handles error reporting in the event that the FPGABUS transfer fails. The test should return false if the oneshot_fpgabus_read/write returns false, otherwise it can continue knowing the transfer succeeded.

k. Every test must call “oneshot_checkquit(state)” at least once every few seconds during testing, and must produce some kind of text output at least this often. Tests that sit around for longer than 5 seconds without any indication that something is happening will not be tolerated. “oneshot_checkquit(state)” checks to see if the user has pressed a key, and if they have it pauses the test and presents a menu with options like ‘quit all tests’, ‘toggle pause on error’, ‘skip to next test’, and so on. The oneshot_checkquit() function does not print anything if no keypress is detected, but this is often a convenient time for the test to print a dot or something to indicate to the user that progress is being made. Calling oneshot_checkquit(state) at least once every few seconds is paramount to making the oneshot test system successful. It gives the user responsive control over the test flow at any time. Tests that do not call this function frequently will not be tolerated.

Following these rules will keep the oneshot test system running smoothly and everybody will enjoy using it. Thank You!!!
5. **EMULIB Library**

Some customers already have their own applications and prefer to make library calls to access the Dini Group hardware without the need to merge the high level Emu code into their app. By simply compiling the “EMULIB” folder into a standalone library this is accomplished.

I. **Methods of using the EMULIB library**

There are two options: compile the EMULIB source directly into your application, or compile the EMULIB source into a library, and then link your application to the library.

The code in the EMULIB folder is independent from the rest of the Emu program, except for the list of Oneshot Test functions for each product. This dependency is only included if the INCLUDE_BOARD_TESTS compiler constant is defined, so as long as this is excluded, the EMULIB folder should compile independently from the rest of the Emu application code. So remember:

**IF INCLUDE_BOARD_TESTS is defined, EMULIB will not function as a stand-alone library.**

The stand-alone library therefore does not support running the pre-written self-test diagnostics on the hardware using the Oneshot Test System- that is what the Emu program is for. Customers are welcome to write their own quick diagnostics using the API functions provided if this type of sanity checking is desired, or they can pull in the rest of the Emu application source code and define INCLUDE_BOARD_TESTS if they really want access to the diagnostic board tests.

II. **Compiling EMULIB directly into an application**

It is recommended to build EMULIB into a library and link to it, following the example in the project_emulib folder. However, without too much effort, an EMULIB application can be built directly off of the EMULIB source code. Follow these guidelines to get started.

First, decide if the AETEST LIBRARY support is needed. The AETEST LIBRARY provides the PCIEDIRECT interface and the FPGA READBACK (aka ReaderBacker) functionality. If these are not needed then the AETEST LIBRARY can be excluded, simplifying the build process.

Include all files in the “emu/App/source/EMULIB” folder in the project you are working with.

If you are using a windows machine, do not include `emulib_os_dep_linux.cpp`
If you are using a linux machine, do not include `emulib_os_dep_win32.cpp`
If you are including the AETEST LIBRARY, then do not include `diniapi_pciedirect_stub.cpp`
If you are not including the AETEST LIBRARY, then do not include `diniapi_pciedirect.cpp`

Make sure that INCLUDE_BOARD_TESTS is not a defined compiler constant.

If building on windows, two external Pcie driver headers are required: “Drivers/windows_pci/GUIDs.h” and “Drivers/windows_pci/loctl.h”. These are located in the “emu/Drivers” folder. If this folder structure is undesirable for your project, copy these headers into the EMULIB folder and change the #include lines in “emulib_os_dep_win32.cpp” to the correct path.

The files “DiniCmos_interface.h” and “DiniCmos_interface.cpp” are also required (applies to both windows and linux). These files are located in “Software/Marvell/DiniCmos_aetest/includes” and
“Software/Marvell/DiniCmos_aetest/sources”, respectively. If this structure is undesirable to your
project, copy the files into the EMULIB folder, and change the #include line in “diniboard.h”. This is the
only file that references this external dependency.

If the AETEST LIBRARY support is included, then “AETest_interface.h” and “AETest_interface.cpp” are
needed. These are located in “Software/Marvell/DiniCmos_aetest/includes” and
“Software/Marvell/DiniCmos_aetest/sources”, respectively.

If the AETEST LIBRARY support is included, then include all headers (*.h) from “AETest/aetest” in your
project. You will need to separately build the AETEST LIBRARY and link it to your application. From QT
you can use the project in Software/emu/App/project_aetest_library to build it. For MSVC you can use
nmake to build the library simply by running “make” at the Visual Studio command prompt in the
Software/AETest/aetest folder. Similarly, in Linux using gcc you can type “make” in the
Software/AETest/aetest folder to build the library.

For windows builds, there are two other library dependencies that must be added to your project if they
are not already used: ws2-32 and setupapi. If you are using QT, simply add these to your ‘.pro’ project
file:

```
win32:LIBS += -lwsock32 -lsetupapi
```

The header file “diniboard.h” must be included in your application to declare all of the API functions
and other features of the library. This file is found at “App/source/EMULIB/diniboard.h”.

III. Building EMULIB into a standalone library

For both Windows and Linux builds you must decide whether AETEST LIBRARY support is required for
your application. The AETEST LIBRARY provides the PCIEDIRECT API interface as well as the FPGA
READBACK “DN Readbacker” functionality. If you do not plan to use these features then it is
recommended to build EMULIB without AETEST LIBRARY support. Edit the QT project file or Makefile
that you are using and there will be instructions for including or excluding the AETEST LIBRARY support.
If AETEST LIBRARY support is included, then the Aetest Library must be built and linked to along with the
EMULIB library. This can be done either with QT using “Software/emu/App/project_aetest_library” or
with gcc/MINGW/MSVC from “Software/AETest/aetest” simply by typing “make”.

Windows
A QT project is provided at “App/project_emulib/emulib.pro“. By default the project uses the project
settings in “App/project_qt/config.pri”. Either set your configuration settings in the config.pri file or
remove the “include ../project_qt/config.pri” line from the emulib.pro file and make the settings and
directed there. Either way, you must NOT use INCLUDE_BOARD_TESTS, and you must choose whether
to use INCLUDE_AETEST_LIBRARY.

Once those two settings are made, open the project in QT Creator and build it. It will produce a file
called “libemulib_win32 DBG.a” which can then be linked to from your Windows applications. Note that
QT uses the MinGW compile environment which uses gcc, therefore the library is in the libx.a format.
The x.lib format is Microsoft specific and is only produced if you are using Microsoft Visual Studio.
A static library will be created, which is desirable in most situations. When linking this will put the EMULIB code directly into your application executable. If for some reason a shared library is required (.dll for windows or .so for linux), there are instructions in the emulib.pro project file for doing so.

QT is not needed to build the library. If gcc is to be used, simply build with the Makefile provided at “App/project_emulib/Makefile.gcc”. The MinGW distribution of the GNU tools works great and is free. Use the command “make –f Makefile.gcc”. This will of course also work in Linux with the real GNU tools.

For Visual Studio, see the section in the “Building Emu” chapter about building in Visual Studio and make the changes discussed there first. Once that is complete, use “App/project_emulib/Makefile.nmake” to build the “emulib.lib” library file. From the appropriate Visual Studio command prompt, use the command “nmake /F Makefile.nmake”.

**Linux**
The QT project in “App/project_emulib/emulib.pro” works for building the library both in Windows and Linux. Make your configuration settings in “App/project_qt/config.pri” or at the top of emulib.pro. You must NOT use INCLUDE_BOARD_TESTS, and you must choose whether to use INCLUDE_AETEST_LIBRARY.

Once those two settings are made, simply open the project file in QT and build it. It will produce “libemulib_linux_dbg.a” which can then be linked to from your Linux application in the standard way (ie –lemulib_linux_dbg).

A static library will be created, which is desirable in most situations. When linking this will put the EMULIB code directly into your application executable. If for some reason a shared library is required (.dll for windows or .so for linux), there are instructions in the emulib.pro project file for doing so.

QT is not required to build the library. The Makefile provided at “App/project_emulib/Makefile.gcc” will build the library with gcc and ar in the standard way. Use the command “make –f Makefile.gcc”.

You can include the library in your application by linking to it in the standard way: -lemulib_linux_dbg. You may also need to add a library search path depending on where you place the libemulib_linux_dbg.a file, which is done with “-L[folder_name]”. Use “-L.” to include the current directory in the library search path.

**IV. Using the EMULIB API**
Outside of this user manual, there are three main sources of documentation for using the API functions: The “EMULIB API SUMMARY.pdf” document, the project_emulib/sample_app, and the Diniboard header file:

emu/Documents/EMULIB API SUMMARY.pdf

This document gives an overview of the available API functions and has some sample code snippets of how to use the API.

emu/App/project_emulib/sample_app

The simple example in “main.cpp” shows all the basics of detecting hardware, connecting to a board, setting clocks, configuring FPGA's, communicating with the FPGABUS, and handling API errors.
This is the main API header file. Start at the BOTTOM of this file, where all of the API functions are listed, with comments describing their usage.

A brief description of how the EMULIB API is to be used follows:

In the file in which you will be calling board functions, #include “diniboard.h”.

EmuLib provides several global variables for convenience. Use these variables to manage the Diniboard objects that are returned by the various API functions:

```c
String EMU_VERSION; // For reporting the version of Emulib
String EMU_DATE; // For reporting the release date of Emulib
Diniapi* global_usb_api; // For discovering boards on USB
Diniapi* global_pcie_api; // For discovering boards on PCIe
Diniapi* global_pciedirect_api; // For discovering boards on PCIEDIRECT
Diniapi* global_ethernet_api; // For discovering boards on Ethernet
list<Diniboard> global_board_list; // For holding the objects for discovered boards
Diniboard* global_selected_board; // For holding a pointer to the active board
```

Emu supports multiple interfaces. At the time of this writing it supports USB, Ethernet, and PCIe, but the framework is extensible and new interfaces may be added in the future. Interfaces are implemented through a base class called Diniapi. Each interface implements a subclass of Diniapi. For example, the class Diniapi_ethernet is derived from Diniapi and implements the Ethernet interface. A global object of each interface is created at program start, and global pointers are provided to the user to access the API’s. For example, to find any boards connected to the USB interface an application would call:

```c
global_usb_api->emu_discover(global_board_list);
```

A list of Diniboard objects is returned describing the available boards. To connect to a board, the application calls the connect method from the associated Diniboard object:

```c
global_selected_board = &(global_board_list[0]); // Or use list iterators to explore the global_board_list
```

Now the global_selected_board can be accessed using any of the Diniboard methods.

**To discover boards:**

```c
global_board_list.clear();
global_usb_api->emu_discover(global_board_list);
global_pcie_api->emu_discover(global_board_list);
global_pciedirect_api->emu_discover(global_board_list);
global_ethernet_api->emu_discover(global_board_list);
```

//`global_board_list` is a global list of Diniboard objects which now contains all found boards.

**To select a specific board:**

/only after discovering boards

```c
list<Diniboard>::iterator iter;
DINIAPI_STATUS apistatus = DINIAPI_STATUS_BOARD_NOT_FOUND;
for(iter = global_board_list.begin(); iter != global_board_list.end(); iter++){
    if(iter->board_info.serial_number == #######) {
        global_selected_board = *iter; // Get a pointer to the matching board if desired, or just use iterators!
```
apistatus = iter->emu_connect();
if (apistatus != DINIAPI_STATUS_SUCCESS) { // Make sure the connection succeeded
    emu_report_systemerror(apistatus);
}
break;
}

// ######## = serial number of the board you want to interact with

To interact with the selected board:
// only after successfully connecting to a board

Now, assuming  apistatus==DINIAPI_STATUS_SUCCESS, global_selected_board points to the board
which you selected (with the serial number #######)
To interact with the board, just call functions from the board
global_selected_board->emu_set_clock(“G0”, “200.0”); // Set clock G0 to 200Mhz

All available functions to be called from global_selected_board are located at the bottom of
“Diniboard.h”

V. EMULIB API Error Reporting

If something goes so terribly wrong that there is nothing to do but give up, then the assert(false) call is
used to force an abnormal abort and give the user a chance to debug the problem. The ‘NDEBUG’
constant is never defined, even in release builds, so that assert() will always cause program termination.
Assert is only used to indicate the program itself is broken, never to handle errors that could occur in
normal applications.

For errors that are slightly less offensive, there is an error reporting mechanism that uses error return
codes supplemented with a text buffer for extra information. This is similar to the “getlasterror()”
mechanism that many programmers are familiar with. Every API function that interacts with the
hardware returns an error code of type DINIAPI_STATUS. Callers should check this status by c
omparing
it to DINIAPI_STATUS_SUCCESS (which is defined to be zero). The macro DINIAPI_STATUS_STRINGS() can be used to get a static string describing any error code.

EMULIB code adds extra error information to a buffer using the following functions:
void emulib_errorput(const string& errortext);
void emulib_errorputf(const char* format, ...);

Higher level code accesses this information by calling:
void emulib_errorget(string& errortext);

Putting this to work, an EMULIB API call might look something like this:

DINIAPI_STATUS apistatus = myboard->emu_clear_fpga(fpga_bitfield);
if (!apistatus) {
    cout << “ERROR: “ << DINIAPI_STATUS_STRINGS(apistatus) << “\n”;
    emulib_errorget(errortext);
    cout << errortext;
    return 1;
}
Because the above code is so common, it is a basic requirement to define a function for it. In the Emu application it is declared as follows in “emu.h”. If using EMULIB as a standalone library, you can use the version supplied in the sample_app “main.cpp” file, or write your own.

```c
void emu_report_systemerror(DINIAPI_STATUS apistatus);
```

Any place in the application where a function that returns type DINIAPI_STATUS is called, it is followed by a check of the return status and a call to emu_report_systemerror() if the status is not DINIAPI_STATUS_SUCCESS (ie non-zero). For example:

```c
apistatus = global_selected_board->emu_clear_fpga(fpga_bitfield);
if (!apistatus) { emu_report_systemerror(apistatus); return 1; }
```
6. **Command Line Emu and Scripting**

In addition to being an interactive menu application, the CMD version of Emu can be used to automate common tasks in scripts or to run simple operations from the command line. The examples below show emu running in linux, but the same operations can be done from the command shell in windows and also using emu_mv on the onboard Marvell processor.

I. **Command Line Menu System**

To enter the standard menu system simply run emu with no command line options:

> emu_cmd_linux

In this mode, commands are run by navigating the menu system and choosing the available options. Each command option prompts the user for the parameters it requires.

II. **Command Line Interpreter**

All of the command options available in the menu system are also available in the command interpreter mode. First, run emu with the –h flag to see a list of available commands:

```
> emu_cmd_linux -h
Emu Version 1.0.9, compiled May 24 2010
Usage: emu_cmd_linux [-<option>+
Where <option> can be one of the following:
   h: print this message
   m: force manual board selection, overrides .ini setting
   c: enter command interpreter after processing options

Command Interpreter accepts commands on stdin.
Parameters match what is asked for if no parameters are given.
When a parameter is a list asked for, use the text of the option or its number.
Replace spaces with underscores when specifying the text of a list option.
The following commands are supported:

   select_board
   specify_board
   select_by_ip
   display_board_info
   set_board_info
   system_call
   reconnect
   disconnect
```

To run the program in command interpreter mode use the –c flag:

> emu_cmd_linux –c

In this mode an EMU>> prompt is given and commands can be entered with or without their parameters. If parameters are not given then the program prompts for them as needed. This mode is most useful for experimenting with the commands to determine what parameters need to be passed in when using them from a script.
Piping commands into emu in command interpreter mode results in the program processing those commands:

```plaintext
> echo set_clock G0 200 | emu_cmd_linux -c

> cat command_file.txt | emu_cmd_linux -c
```

The first example will only work if Emu has previously been run and connected to the target board, thus saving the target board’s information in the emu.ini file for auto-connection when Emu starts up. Otherwise a board connection command needs to be given before any commands. This can easily be done in a command file:

```plaintext
//begin command_file.txt
specify_board 1004006 ethernet
set_clock G0 200
//end command_file.txt

> cat command_file.txt | emu_cmd_linux -c -m
```

This results in the G0 clock being set to 200Mhz using the Ethernet interface on the board with serial number 1004006. The `-m` flag turns off the auto-connect feature, which we don’t want because the first thing we do is connect to a specific board.

Note that when any command has you pick an item from a list, you can pass the number of the list item as the parameter OR you can pass the text of the list item with spaces replaced by underscores. This is useful when the number of the list item may change across program runs but you want your script to always select the same item. For example “configure_fpga_from_host FPGA_B fpga_b.bit” will always configure FPGA B with the bitfile, whereas “configure_fpga_from_host 3 fpga_b.bit” will only work if FPGA A is stuffed on the current board. On a board without FPGA A, menu item 3 will be FPGA C!

Because parameters are whitespace delineated, it is a problem if you need to pass in a parameter that contains spaces. To solve this problem emu parses the input stream for backslash escaped spaces and doesn’t break parameters on those boundaries. An example is given below:

```plaintext
EMU>> system_call echo hello world!  WRONG!

EMU>> system_call echo\ hello\ world!
```

The system_call command takes one parameter, which is sent to the Marvell processor and run as a system call. In the first example emu will interpret the input as a command followed by 3 parameters and only the first one, “echo”, will be given to the “system_call” command. The other two parameters, “hello”, and “world!”, will remain on the parameter stack for other commands- which is NOT what we wanted. The second example correctly escapes the spaces with backslashes resulting in one parameter.

If you need a backslash space combo in your input text, then escape the backslash with a backslash and the space with another backslash:

```plaintext
EMU>> system_call echo\ this\ is\ a\ backslash\ followed\ by\ a\ space:\\\!
```
Ugly yes, but this allows you to get any combination of backslashes and spaces that you may require. What will really make your brain hurt is that in the above example, echo interprets the backslash space combo we sent as an escaped space character and doesn’t print the backslash!

The chapter “Using Emu On The Marvell” discusses further using emu_mv in command interpreter mode in order to auto-configure boards from various places using the provided dini.sh shell script and config.txt command file. Continue to this chapter for more details.
7. The Onboard Marvell Processor

Emu can’t do anything without some firmware on the other side for it to connect to. This chapter describes the firmware environment and gives tips for those who want to develop their own Marvell applications to run on the board.

I. The U-Boot Boot Loader

When a Dini Group board with onboard Marvell Processor is powered on, the processor wakes up and begins executing code. Specifically, it executes code out of a SPI Flash device that has been factory programmed with a customized boot loader called “U-Boot”. The boot loader code is rarely, if ever, changed, and firmware updates do not update it. It is loaded onto the board by the factory using a SPI Flash programmer plugged into the appropriate connector on the board.

U-Boot turns on and configures basic peripherals and then pauses for a few seconds to allow users to break into its command shell. The Marvell RS232 port, sometimes labeled “RS232 CPU”, is used for interaction. Connect to the port with the provided serial cable adapter at 19200bps, no parity, no flow control. We recommend “putty” as an appropriate and free terminal program (google “putty” to get it).

If a keystroke is not received, or if the “boot” command is given from the U-Boot command shell, then a normal boot commences. U-Boot copies the Linux kernel from the SPI Flash into DRAM and then jumps to it.

If a keystroke is received, then the U-Boot command shell is entered. Type “help” for a list of commands. The most common operation done here is “run spi_boot_recoveryfs”, which boots into the recovery partition for performing a firmware upgrade. See the chapter on upgrades for more information.

Another common operation is to modify U-Boot environment variables. Type “printenv” to see a list of the U-Boot environment variables. Some variables of particular interest are “ethaddr” which stores the boards Ethernet MAC address, and “rw_val” which specifies how the root filesystem is mounted in a normal boot (ro=read only, rw=read/write). The “bootcmd” environment variable contains the text that is executed when the “boot” command is given (or when no keystroke is received to break into the U-Boot command shell). Use “setenv [variable] [value]” to change an environment variable. Use single quotes around [value] if it contains spaces. Use “saveenv” to write new values out to the SPI Flash to preserve them across power cycles. Type “boot” to continue the boot process with the current environment settings. If environment variables are changed but not saved with “saveenv”, then the board will boot with the modified environment, but will revert to the original settings on the next power cycle.

II. The Linux Kernel

```
-sh-3.2# uname -s -r -v -m
Linux 2.6.22.18 #301 Tue Oct 28 09:33:55 PDT 2014 armv5tel
```

The Linux kernel was originally provided by Marvell and has been patched by us for correct operation in our hardware environment. Linux boots up and finishes configuring peripherals, mounting the root file system, and running the startup script at /startup.sh.
III. The Root Filesystem

In a normal boot sequence Linux mounts the root filesystem which is located on a NAND Flash device. This root filesystem is what gets updated when a firmware update is run; the entire root filesystem is removed and the new version is installed. The root filesystem contains the startup scripts, the DiniCmos firmware program, the ConfigFPGA programming file [Note: newer boards store the ConfigFPGA programming file in a separate SPI Flash device], as well as most of the standard Linux utilities you’d expect.

The root filesystem is mounted read-only by default. This prevents accidental deletion, and also prevents filesystem corruption when the board is powered off without first notifying Linux (ie by issuing the “halt” command). It is highly recommended that users never modify the root filesystem directly; a “user” partition is provided for customer modifications. A U-Boot environment variable called “rw_val” controls this. Break into the U-Boot command shell and change “rw_val” to “rw” for read/write or “ro” for read only. See the U-Boot section above for details.

IV. The startup.sh Script

When Linux finishes booting up it launches the startup.sh script. To view this script on the Marvell terminal, type “more /startup.sh” at the command prompt. This script does some final setup, including configuring the Ethernet connection (using a static IP if it’s set up, or launching udhcpc to acquire an IP address using DHCP). Finally, it starts up the DiniCmos program which runs in the background.

V. DiniCmos

The firmware program that provides the Ethernet, USB, and PCIe connections for Emu is called “DiniCmos” and is installed in the root file system at:

/Dini/Software/Marvell/DiniCmos_aetest/out/DiniCmos

When the program is run, it configures the ConfigFPGA with the programming file pointed to by the Board Info Partition (unless it was already configured from SPI FLASH). It then loads the various device drivers, launches threads for monitoring the various interfaces, and does some final board setup. Finally, it prints out its version number and a message declaring that it is ready to accept host connections.

VI. Filesystem Partitions

The NAND Flash is divided into 5 partitions. Type “Is /dev/partition*” on the Marvell terminal to see them: partition_root, partition_boardinfo, partition_recov, partition_user, and partition_spi. It is highly recommended that only “partition_user” is modified by customers.

The root partition has already been discussed.

The boardinfo partition is a small read/write partition used for storing data about the board such as what model it is, which FPGAs are installed and of what type, and the location of the ConfigFPGA programming file. (Note: on newer boards this partition only holds information about the onboard CPU, and information about the FPGA board is stored in an external EEPROM device connected to the FPGA board’s ConfigFPGA) The information in the boardinfo_partition is preprogrammed by the factory before each board is shipped. When Emu issues the “Display Board Info” command, DiniCmos responds mostly with information that is stored in this partition. The “Set Board Info” command in Emu is used to write the board information into this partition.
The **reco**v partition (short for “recovery”) contains a minimal file system that is capable of downloading a root filesystem image from DiniGroup servers and installing it. In the firmware update procedure, this is the partition that is mounted when the “run spi_boot_recoveryfs” command is given from U-Boot.

The **user** partition consists of the remaining space in the NAND Flash and can be used for anything, including setting clocks and configuring FPGAs at system boot (see the chapter on “Configuring from the onboard NAND Flash” for details). It is empty when the board ships and may be used for storage of user code and applications or for data to be used in a custom application. By default, the startup.sh script mounts the user partition at “/mnt/partition_user” at system startup. Do “cd /mnt/partition_user” and you are in a non-volatile read/write space to play with. Note that because Linux buffers filesystem I/O operations it is not safe to simply power off the board when working with a writeable filesystem. Use the “sync” command to flush all I/O operations, or use the “halt” command to shut down the processor before disconnecting power. Unmounting the partition (umount /mnt/partition_user) will also force all buffered I/O to flush. Ignoring this practice WILL result in data loss whenever data is written to the user partition.

The **spi** partition maps to the SPI Flash where U-Boot and the kernel image are stored. This read-only partition is typically not useful for customers, but could be used to write a new U-Boot image into the SPI-Flash without using an external SPI Flash programmer.

It is also worth noting that DRAM space can be used for a ram-disk filesystem, which can be mounted at “/mnt/ram” using the command “sh /mnt/ram/mount_ram.sh”. Remember this is volatile memory and anything stored here will be lost when the board is powered down, or when the ramdisk is unmounted.

### VII. Developing on the onboard Marvell CPU

After DiniCmos finishes initializing, hitting “enter” on the Marvell terminal will display a Linux command prompt. This is a Linux bash shell. Most of the common Linux utilities are present, provided by “Busybox” (see [www.busybox.net/about.html](http://www.busybox.net/about.html) for more information). By default the filesystem is mounted read-only, see section I of this chapter for more details.

Status messages are periodically displayed on the Marvell terminal which can be distracting if the terminal is being used for real work. Instead, connect the board to Ethernet and telnet in from your host machine. Username is “root” and Password is “root”. To find the hostname to which you can telnet, type “hostname” at the Marvell terminal. To get the IP Address to which you can telnet, type “ifconfig” at the Marvell terminal. You can also get the board’s hostname and IP address from Emu by using the “Board->Display Board Info” command. See previous chapters for information on setting up the Ethernet connection with DHCP or Static IP address.

Editing source code on the Marvell terminal or in a telnet session can be very cumbersome. Instead, mount your workstation’s hard drive and do all of your editing right on your workstation:

```
-sh-3.2# mount -t cifs //[machine]/[share_name] /mnt/dncvs -o username=user,password=pass
```

This command mounts a windows share (aka samba share) to /mnt/dncvs in the root filesystem. You can then “cd” into your workstation’s filesystem in order to compile and run the firmware you are developing. This is the recommended method of developing code for the onboard Marvell CPU. It gives the ease and flexibility of editing source code on the workstation with any text editor, but retains the
simplicity of not requiring a cross-compiler or having to move the binaries anywhere in order to run them.

Standard gnu “gcc” and “g++” compilers are available for building user programs. The target architecture is “arm-none-linux-gnueabi”; “/usr/bin/arm-none-linux-gnueabi-g++” is the same as “/usr/bin/g++”. See the “Makefile” for the DiniCmos program for an example of how to build an application on the Marvell.

A cross-compiler does exist and can be used to build on a workstation instead of using the Marvell cpu directly. We recommend using the methods above to natively build user applications for the Marvell processor, as there isn’t much downside of doing so. However, to build the kernel itself the cross-compiler is required. Therefore details about setting up a cross-compile environment are included in the section about the kernel below.

VIII. U-Boot and Kernel Modifications

Modification to U-boot, the kernel, and the root filesystem are not supported by Dini Group. It is highly recommended that only the “partition_user” partition is ever modified by customers in the field.

The U-boot and kernel source can be obtained by contacting support@dinigroup.com. There is no package manager and any kernel changes require that the kernel is recompiled from source. The root filesystem is available for download from the firmware update server- see the instructions on doing a USB Flash Drive firmware update for details on downloading the root filesystem image. The image can be uncompressed and modified before installing it on the NAND Flash, or it can be modified in place by mounting the root filesystem in read/write mode (see the section above on U-Boot for details).

To compile the U-boot and kernel source code the Marvell SDK should be obtained. Contact Marvell support for more information. Failing that, the cross-compiler that ships in the Marvell SDK is freely available and below we describe roughly how to get it working. Attempting to do native builds using the Marvell CPU is not recommended, as some required tools such as perl and kbuild are missing.

At the time of this writing, the cross-compiler packages could be freely downloaded at the links below. Send an email to support@dinigroup.com if these links do not work anymore.

32-bit linux installer:

32-bit windows installer:

32-bit linux tarball:

32-bit windows tarball:

source tarball:
We recommend getting the linux installer package and running it on your target system. We have tested the linux installer in Fedora 10 x86 with success, and use this environment to do our builds. We found while working with Marvell that Fedora linux seemed to be the distribution best supported by them, even though the SDK claims support for other kernels with no mention of Fedora! We have also successfully installed the Windows version and built a simple application with it that runs correctly on the Marvell processor. The Windows version is probably ok for application development but for building the Kernel and U-Boot, using the linux version on an x86 version of linux is required.

Both the U-boot and the kernel source that we provide have Makefiles for building. With the cross-compiler set up, simply type “make” in the top level folder of U-boot, or “make ulmage” in the top level folder of the kernel source to build the targets (u-boot-db78200_MP.bin and ulmage respectively). In the kernel source, typing “make menuconfig” brings up a menu for customizing the kernel. The “mkimage” tool is required for the kernel build to format the output into the ulmage format. If it is not present, the message “"mkimage" command not found - U-Boot images will not be built” is displayed at the end of the build. This tool can be installed using the package manager on most linux distributions, however it is probably simpler to use the one from the U-boot package (Building U-boot will also build the mkimage tool and place it in the “tools” folder). Put “mkimage” in your path and then your kernel build will complete.

There are several options for booting a kernel and mounting a filesystem. Dini Group boards by default will boot the kernel from the onboard SPI flash and mount the root filesystem in the onboard NAND flash. The default method of the Marvell SDK is to boot the kernel from a network server using tftp, and nfs-mount a root filesystem on the network. This latter method is preferable for development. There are U-boot environment variables set up for many combinations of these. Type “printenv” at the U-boot command prompt to examine them:

- **bootcmd**: This gets run when you type "boot" at the U-boot prompt, we set it to "run spi_boot_nandfs"
  
  This is also run if the board is allowed to boot on its own, without breaking into U-boot.

- **spi_boot_nandfs**: Boot kernel from SPI flash
  
  Mount root filesystem in NAND Flash

- **spi_boot_netfs**: Boot kernel from SPI flash
  
  Mount filesystem on nfs-mounted server"serverip/rootpath"

- **net_boot**: Boot kernel using tftp from "serverip/image_name"

  Mount filesystem on nfs-mounted server at "serverip/rootpath"

- **spi_boot_recoveryfs**: Boot kernel from SPI flash

  Mount recovery filesystem from NAND Flash

To execute any of these, type “run [var]” at the U-boot command prompt, for example “run net_boot” would execute the commands in the “net_boot” environment variable. The behavior of most of these is controlled by other environment variables, such as “serverip”, which specifies the ip address of the network server to connect to. See the section about U-boot above for instructions on modifying environment variables. By setting these variables correctly for your environment, and using “saveenv” to store them into the onboard SPI flash, the existing commands can be made useful for any setup.
Note that a command called “net_boot_nandfs” is missing; this command would be useful for kernel developers that don’t want to load their kernel builds into the spi flash before testing them out, and don’t want to set up an NFS Mount in order to use the “net_boot” command. This command can be created as follows:

```
dini_uboot>> setenv net_boot_nandfs 'tftpboot 0x2000000 $(image_name); setenv bootargs $(console) root=/dev/mtdblock3 $(rw_val) cpu0=$(cpu0_res) cpu1=$(cpu1_res); bootm 0x2000000;'
dini_uboot>> saveenv
```

Then use “run net_boot_nandfs” to load the kernel image from the TFTP server and boot it, mounting the regular NAND root filesystem.

**Kernel Developers: Installing New Kernel Images**

To install a new kernel image into the onboard SPI flash, replacing the currently installed kernel, we use the “copy_kernel_to_flash” U-boot variable. This command loads the kernel image into DRAM from a tftp server, and then programs the image into the SPI flash. Type “printenv” from the U-boot prompt to see the definition of “copy_kernel_to_flash”. The ip address of the tftp server is specified by the “serverip” environment variable (our tftp server is at 192.168.2.41 on our internal network). The IP address used by the board is specified by the “ipaddr” environment variable. Be sure this does not conflict with anything else on your network. Obviously an Ethernet cable must connect the board to the network containing a machine that is running a tftp server at the ‘serverip’ address. The tftp server must contain the kernel image at the location specified by the “image_name” environment variable (MV78200/uImage by default). Type “run copy_kernel_to_flash” from the U-boot commandline to install the kernel into the flash. When the command completes, type “boot” to boot into the kernel, or power cycle the board.

**U-Boot Developers: Installing New U-Boot Images**

To install a new u-boot image onto the boot flash, the dini_uboot_updater can be used as described in the chapter “Upgrading Software and Firmware”. However, if a new build fails for some reason it will render the board unusable, and you will not be able to reach a linux prompt in order to run the dini_uboot_updater again. Therefore, u-boot developers will need to obtain the flash programming cable: SF-100 from Dediprog Engineering. Use the flying-leads attachment to plug this cable into the 5-pin programming header on the Dini Group board. The board must be held in hard reset during programming cable operation, which requires holding down the board’s reset button while the Dediprog software is running. The instructions for using the SF-100 cable are below:

1. Connect the SF-100 SPI Programmer to the SPI programmer header on the board. Consult the board’s user manual, schematic, or support@dinigroup.com for help.
2. Turn on the board.
3. In Windows, run the "Dediprog Engineering" software.
4. In the Dediprog program:
   a. Hold down “Hard Reset” button on the board and click "Detect chip".
   i. Select M25P128 from the menu.
      (If you do not HOLD DOWN reset button, it will say "no device detected")
   b. Click the "File" button.
      i. Select the u-boot image file. (like “u-boot-db78200_MP.bin”)

c. Click the "config" button.
   i. Choose "Update memory only on sector locations with content difference."
   ii. Choose "Update start from address (Hex) and enter "F60000" in the text box.
d. Hold down “Hard Reset” button on the board and click Batch. The "batch" operation takes about 10 seconds. It will tell you when it's done. HOLD DOWN_RESET BUTTON UNTIL IT IS COMPLETE.
e. Hold down “Hard Reset” button on the board and click Verify. Make sure it succeeds.
f. Turn off the board and remove the SPI programmer cable from the board.

IX. Setting the Root Password
By default the root password for the Marvell linux install is “root”. If required, this can be changed by following this procedure:

1. Boot OS with filesystem in read/write mode (See the section “The Root Filesystem” for details)
2. cp /etc/passwd /etc/shadow
3. passwd (Follow prompts to set your new password)
4. cp /etc/shadow /etc/passwd
5. Reboot (Do NOT power the board off without doing a reboot or halt command)

The new password will take effect immediately after step 4. Step 5 is necessary to flush the filesystem writes out to the flash to keep the filesystem coherent.

X. Installing Alternate Operating Systems
It may be possible to install and run alternative operating systems, such as VxWorks or real-time linux, on the Marvell processor but this usage is far beyond the scope of this manual and beyond the scope of what Dini Group can directly support. Consult the Marvell SDK and Marvell support for further information on kernels, operating systems, boot loaders, and filesystems.
8. Using Emu On The Marvell

The onboard Marvell processor runs an embedded linux operating system, which takes about 30 seconds to boot up. This is why it takes about 30 seconds after the board is powered on before Emu can detect it. By connecting a serial port cable to the “RS232 CPU” port on the board, a terminal program can be connected to get a linux shell. (We recommend Putty which is freely downloadable. Use terminal settings: 19200bps, no parity, no flow control). If Ethernet is connected to the board you can also ssh or telnet in once the board has acquired an address over DHCP (or by using the static IP address if this option has been enabled). Username is “root”, Password is “root”. If using DHCP, Emu will tell you the IP address that the board got when you use the “Select Board” option, or “Display Board Info” option. The boot partition is set to “read-only” mode, so that nothing gets corrupted when users turn off the board without properly “shutting down” linux. If you wish to do development on the Marvell and need more information about the kernel installation, uboot boot-loader, and DiniCmos application see the chapter “The Onboard Marvell Processor”.

I. Configuring from a USB Flash-Drive or SATA Hard Disk

The CMD version of EMU compiles for the onboard Marvell CPU, and comes preloaded on the NAND filesystem as “emu_mv”. This provides the flexibility for the Marvell processor to interact with the board without intervention from a host machine. The most common use for this feature is for setting up clocks and configuring FPGA’s at startup from data stored on a USB flash-drive or SATA hard disk.

When a USB flash-drive is connected to one of the USB “A” type connectors on supported Dini boards or a SATA hard disk to one of the SATA ports, the Marvell processor automatically mounts the drive and, if present, executes the shell script “dini.sh” from the root of the drive. The Marvell uses the linux “hotplug” mechanism in which hotplug events are sent to a user mode script located at /sbin/hotplug. The script parses the events and when a USB Flash Drive or SATA hard disk is connected to the system attempts to mount it. Most drives will attach to /dev/sda, and the hotplug script will then mount them at /mnt/sda. Some drives will attach to /dev/sda1, in which case the hotplug script will mount them at /mnt/sda1. If a second drive is connected, it will attach to /dev/sdb or /dev/sdb1 and be mounted at /mnt/sdb or /mnt/sdb1 respectively. No matter where the drive ends up being mounted, the hotplug script will execute “dini.sh” from the root of the drive after it is mounted, if the file exists.

Users are free to modify the “dini.sh” script to do whatever they like, but the default version will be fine for all but the most advanced users. Most users will only want to edit the “config.txt” file, which is discussed below, but first take a look at the provided sample dini.sh script:

```
Software/emu/Sample_Scripts/dini.sh
```

The script attempts to connect to the local board, and if it can’t it waits a bit before continuing. This protects against the case where the USB flash drive came up before the DiniCmos software was launched on the board. The emu_mv installed on the Marvell will connect to the local board using the Ethernet loopback adapter (address 127.0.0.1) by default when it is started (ie as long as the –m flag is not passed). Although it is using the Ethernet interface, data will not go out of the Ethernet port when we communicate with the local board.
We then send the contents of a text file called “config.txt” to emu. Config.txt is simply a list of emu commands and parameters. Emu will quit when it receives an EOF on stdin, so explicitly sending the “q” command is not necessary, but you may, if you wish, include the “q” command at the end of the config.txt file to explicitly quit emu. Note the use of dos2unix to force unix style newlines. Finally, the script scans the output of emu for errors and reports any that it finds. This provides a basic example of how to interact with emu in a script: pipe in commands, capture and parse the output.

Most users will only edit the config.txt file to get the clock settings and other features that they want. However, advanced users will find that they can create dini.sh shell scripts that automate their entire test suites doing everything from configuring FPGA’s to writing and reading onboard DRAM.

Now take a look at the “config.txt” file:

```
Software/emu/Sample_Scripts/config.txt
```

This example shows only the most basic emu commands. You can explore the available commands by running “emu_mv -h” at the Marvell processor command-line. To discover the parameters taken by each available command, experiment by running “emu_mv -c” to enter command interpreter mode. You can then enter any available command and see what parameters it asks for. Note that when selecting an item from a list, you may use the list number OR the actual text of the list item, which may be preferred in the case where the list numbers may change across boards with different stuffing options. If list options contain spaces, then replace the spaces with underscores in your parameter text.

Most Dini Group boards ship with a USB flash-drive that already contains these sample scripts in the root folder, ready for your use. If not, simply copy the samples from the locations mentioned above into the root of the provided USB flash-drive and you’re ready to go.

**NOTE:** A link to Emu compiled for the Marvell (called “emu_mv”) is in the /bin folder on the NAND filesystem. This guarantees it will always be in your path. If you wish to use your own version of emu_mv, you can place the binary on the USB flash drive or SATA hard disk and in dini.sh change the references to “emu_mv” to “./emu_mv”, which will force it to run the local copy instead of the copy installed on the system.

**NOTE:** In 2016, we started having the DiniCmos output logged onto any USB Flash Drive that is plugged in to the system. This will not interfere with the execution of the dini.sh script in any way, but you will notice log files being created on the USB Flash Drive. These logs are extremely useful for diagnosing any hardware issues that pop up during the course of a project.

### II. Fixing An Unreadable USB Flash Drive

Some USB Flash Drives come formatted in a way that the Marvell linux distribution can’t read. The provided Dini Group USB Flash Drives will always work, so try these first if your own flash drive is not working. To “fix” an uncooperative 3rd party USB flash drive follow these steps:

1. Connect a terminal to the Marvell RS232 port (19200bps, no parity, no flow control)
2. Boot up the board and when it’s done press enter on the terminal to get the linux prompt.
3. Connect the USB flash drive to the board. If the board has more than one USB port either one is fine.
4. Type the command “cat /dev/zero > /dev/sda” (Erases all contents of the drive)
   
   **Note:** Some USB Flash Drives may appear as /dev/sda1 instead of /dev/sda.
5. Wait until it says “cat: write error: No space left on device” (30sec to 5min depending on size of drive)
6. Remove the USB flash drive from the board and connect it to a Windows PC
7. Format the USB flash drive from the Windows PC as “FAT32”

III. Configuring from the onboard NAND Flash

NOTE: USB Flash Drive and SATA Hard Disks take precedence over the NAND Flash. If a dini.sh script is detected on a USB Flash Drive or SATA Hard Disk then any dini.sh that is present in “partition_user” will not be run. This behavior can be modified in “startup.sh” in the root of the Marvell filesystem.

Sometimes it is desirable to have default clock settings and even default user FPGA bitfiles loaded at startup without a USB flash drive or SATA hard disk hanging off the side of the board. There is 100MB of storage available for customer use on the Marvell processor’s NAND Flash filesystem in an area called the user partition. This space can be used for any purpose (see the chapter, “The Onboard Marvell Processor” for more details), but is conveniently set up for loading user settings exactly like a USB flash drive or SATA hard disk. Read the section above about configuring with a USB flash drive or SATA hard disk before continuing with this section.

After the Marvell processor boots up, it looks for a “dini.sh” script to run. It first checks for USB Flash Drives and SATA Hard Disks as discussed above, and if no “dini.sh” scripts are found, then it checks in the user partition, which is located at /mnt/partition_user. This provides an easy way to set clocks and configure user FPGA’s directly from the onboard NAND Flash. There are many ways to get your settings into the user partition on the NAND Flash, the following procedure illustrates using a USB flash drive to transfer the files from your workstation to the board.

Procedure for copying settings and bitfiles to the NAND Flash user partition:

1. Get a USB Flash drive and connect it to your computer
2. Make sure there is no "dini.sh" file in the root, or it will be run when the flash drive is connected to the board.
3. Make a folder called "tmp" or something, so that the dini.sh that we copy in won't be run when we plug the flash drive into the board.
4. Copy "dini.sh" and "config.txt" in the "tmp" folder you just created on the USB flash drive.
5. Copy your bitfiles into the "tmp" folder. You have 100MB in the user partition, so use bitfile compression if necessary.
   Make sure the filenames match exactly between what is called out in "config.txt" and the actual bitfile names- linux is case sensitive!
6. Connect a terminal to the Marvell RS232 port (See the chapter “The Onboard Marvell Processor” for details).
7. tail -30 startup.sh
   Boards shipped before March 1, 2012 don't have firmware support for the user partition.
   If the last few lines of startup.sh don't mention mounting partition_user and trying to run dini.sh from partition_user, then you'll have to update startup.sh.
   See below for the procedure for updating startup.sh, then come back and continue from here.
8. Safely remove the flash drive from your computer and plug it into the board.
9. The board will auto-mount the flash drive and you will eventually see a line like this:
   Hotplug(254): Device Sucedfully Mounted At /mnt/sda
   It may not be "/mnt/sda" but where ever it is, this is where the files you want to copy are located.
10. cp /mnt/sda/tmp/*.* /mnt/partition_user   [Substitute /mnt/sda with whereever the drive was mounted in the previous step]
    If you put a bunch of huge bitfiles on there this could take a minute or two to complete, so be patient!
11. Halt
    This flushes all writes to partition_user.  NOTE: DATA LOSS MAY OCCUR IF YOU POWER DOWN THE SYSTEM WITHOUT ISSUING THE HALT COMMAND.
12. When the system is down, power cycle the board.  Allow it to boot normally.
13. After dinicmos runs, the dini.sh script that we put into partition_user should be executed and the commands in "config.txt" will be run.
    Your clock settings and FPGA loads are now done by default every time you power on the board!
14. To stop your setting from happening, simply get on the Marvell terminal and do:
    mv /mnt/partition_user/dini.sh /mnt/partition_user/dini.sh.disabled
    halt
    Then power cycle the system and you are back to normal!

Procedure for updating startup.sh on boards shipped prior to 3/1/2012:
There are two options for updating startup.sh.  The first is to run a firmware upgrade.  The latest firmware has the required changes in startup.sh for running dini.sh from partition_user.  The second is to request the latest startup.sh from support@dinigroup.com and replace the old one with the new one as follows:

1. Copy the new startup.sh onto your USB flash drive and plug it into the board.
2. Power on the board with a terminal open to the Marvell serial port
3. Press a key to break into U-Boot when it prompts near the beginning of the boot sequence.
4. setenv rw_val rw
5. boot
6. Let the system boot.  After it says "ready for host connections" or something like that, press enter to get a linux prompt.
7. mv startup.sh startup.sh.bak   [just in case!]
8. cp /mnt/sda/startup.sh .
9. halt

IV. Configuring From A Network Share
It is also possible to set clocks and configure FPGAs at bootup directly from a network share.  This can be useful if several boards are deployed on a network for development, and the FPGA designer wants to periodically update their configurations.  Simply drop the new configuration files on the network share and power cycle all of the boards!

Setting this up requires adding the IP Address, username, and password for the network share into “startup.sh” in the root of the Marvell filesystem.  The steps in the previous section describe booting
with the filesystem in read/write mode. The “vi” editor can then be used to edit “startup.sh”. Search for “ip_address” and fill in the appropriate values for your network share. Issue a “halt” command, then power cycle the card for the new settings to take effect. Contact support@dinigroup.com if assistance is required in setting this up.

Note that if an IP Address is added to “startup.sh” then a networked dini.sh will take precedence over a “partition_user” dini.sh, but USB Flash Drives and SATA Hard Disks will take precedence over the networked dini.sh, and will prevent it from running if present.

V. Debugging Flash-Drive and User Partition Configuration
If flash-drive configuration fails, there is no immediate feedback which can leave the user feeling helpless. Here is a list of things to do before contacting support@dinigroup.com:

1. Make all filenames lowercase, and make sure the commands in config.txt reference the filenames as all lowercase. Linux is case sensitive, while Windows is not, and sometimes filenames that have uppercase letters end up lowercase when copied to the flash-drive on a Windows machine. We therefore highly recommend to use only lowercase letters in filenames. In addition, ‘dini.sh’ and ‘config.txt’ must be all lowercase or they won’t be found.
2. Verify Unix-style line endings in all text files (ie dini.sh and config.txt). The Marvell attempts to run ‘dos2unix’ on all of the files before using them to avoid newline conflicts, but in the event this is not working properly it doesn’t hurt to convert line endings manually.
3. Connect a terminal to the Marvell RS232 port (19200bps, no parity, no flow control). With the terminal connected you have a linux shell to the Marvell processor. Plug in the USB flash-drive and observe the text that is displayed. Look for the text “*** Running dini.sh ***”, to see if it gets to the point of running the shell script or not.
4. The output of dini.sh is logged to a file called “dini.out” on the USB flash-drive. If this file was not created, then dini.sh did not run and there is a problem with the Marvell USB hotplug system. If it was created, it will likely contain error messages pointing to the problem.

If these steps do not solve the problem, then contact support@dinigroup.com and include the text that displayed on the Marvell RS232 port when the USB flash-drive was connected, along with the dini.sh and config.txt files that are on the USB flash-drive.
9. **PCIEDIRECT ConfigFPGA Cable Connection**

(High Speed Host-to-FPGA Data Transfer)

Most of the Marvell based Dini Group products, including all Virtex-7, Virtex-Ultrascale, and Stratix-5 based boards, support the option of connecting PCIe directly to the ConfigFPGA to gain high performance throughput between the Host PC and the User FPGA’s by taking the Marvell processor out of the datapath. The FPGABUS (NMB or TMB) to the user FPGAs is shared between the Marvell access points and the PCIEDIRECT port. It is up to the user to prevent simultaneous accesses from both the Marvell and the PCIEDIRECT, which is not supported. Both ports can be used together as long as the accesses do not overlap.

Typical speeds normally seen using PCIe through the Marvell CPU to the User FPGA’s are 20-30MB/s. With the PCIEDIRECT direct cable connection, speeds around 700MB/s are typical on boards using TMB (Virtex-Ultrascale, Stratix-10, and newer), or about 600MB/s for boards with 100Mhz NMB clock (Virtex-7 and Stratix-5 boards), or 380MB/s with a 50Mhz NMB clock (Virtex-6). This chapter describes how to set up and test the direct cable connection.

I. **Setting up PCIEDIRECT**

When using the PCIEDIRECT port, the host must load the AETest device driver which is distributed inside the Emu release package. The AETest software can be used directly to connect through the PCIEDIRECT to perform basic data transfer operations using the AETest API (see the AETest Manual for details). The AETest software also builds into a library for linking into custom applications. Emu links in the AETest library to implement its PCIEDIRECT access, and implements the EMULIB API on top of it. Using the EMULIB API gives you access to configuring FPGAs, setting up clock networks, issuing logic resets, etc. in addition to simple data transfer.

To verify that the PCI Express Cable interface is working properly, follow these steps:

1. Attach the PCI Express cable between the Dini board and the host PC. On the DN2076K10, the ConfigFPGA cable connector is marked J6.
2. Boot the Dini board. The board is fully loaded when the configFPGA done LED turns on (Blue) and the configFPGA status LEDs begin to blink. Many boards additionally have a “link ready” GREEN LED that turns on once the board is ready to accept host connections.
3. Boot the host PC. If the host PC and the Dini board are using the same power supply and the host PC was booted in the previous step, then reboot the host PC now without turning off the power supply. The host PC must start up with the Dini board already configured so that it can detect the PCIe endpoint in the configFPGA.
4. The device should now show up in Windows Device Manager or on Linux by running:
   'lspci -d 17df: '

To test the speed of the interface:

1. Verify that the interface is working properly (see above)
2. Using EMU, set clock G2 to 200MHz. (Required for Virtex-6 boards only)
3. Using EMU, load the Maintest bitfile into FPGA A
4. Download the AETest software 
   (http://www.dinigroup.com/files/web_packs/Aetest.zip)
5. Load the driver. On Windows, this is located at AETest/wdmdrv/drv/bin. On Linux, go to 
   AETest/linuxdrv-2.6/, run 'make', and then 'sh dndev_load.sh' to build and load the driver.
6. Run AETest. On Windows, run AETest/aetest/aetest_wdm.exe. On Linux, go to AETest/aetest, 
   run 'make', and then run './aetest_linux'.
7. The board should be recognized and reported as a “PCI Express Cable” board. On the 
   DN2076K10 it will report "DN2076K10 Virtex6 PCI Express Cable 4 lane".
8. Press any key to continue to the main menu. At the main menu, select 
   “7) DMA Test Menu”, and then options 5,6, or 7 to test the DMA bandwidth. 
   Option 5 will test host->board bandwidth, option 6 will test board->host bandwidth, and option 
   7 will test host<->board full-duplex bandwidth.

For NMB boards, the maximum bandwidth achievable is either 50MHz * 64bit = 400MB/s, or 100Mhz * 
64bit = 800MB/s depending on the NMB bus speed (V7 and Stratix-5 boards use 100Mhz). For TMB 
boards the transceiver links can transfer about 4GB/s with protocol overhead, but endpoint modules will 
typically use a 64-bit bus at 100Mhz reducing this available bandwidth to about 750MB/s. Typical host 
machines will lose around 10% of the bandwidth to protocol overhead and other inefficiencies so 
expected real-world bandwidths typically seen are in the range of 700MB/s (TMB or 100Mhz NMB) or 
350MB/s (50Mhz NMB). If an application requires bandwidth in excess of 700MB/s contact 
support@dinigroup.com to learn about firmware options for getting closer to the 4GB/s maximum.
10. Board Stacking

Connecting multiple boards together into a single system can greatly expand the FPGA resources available to a design. As demand for multiple-board systems has grown, we’ve developed a system to connect multiple-boards together and host them from a single “master” board. This allows all of the EMULIB API commands to operate on any board in the stack while having a host connection only to the “master” board.

| Virtex-6, Stratix-5, and earlier boards | do not support board stacking. |
| Virtex-7 boards (excluding DNV7F1A) | support board stacking with the DNNMB_CONNECT board. |
| Virtex-Ultrascale and newer boards | support DNBC board stacking. |

I. Four components of a Board Stack

There are four components required to achieve a fully integrated board stack: Host Communication, System Clocking, YMB Interconnect, and FPGA-to-FPGA Interconnect.

FPGA-to-FPGA Interconnect is perhaps the most critical component of a board stack. This refers to point-to-point routes between I/O pins of user FPGAs on separate boards.

YMB Interconnect refers to multi-point nets that connect between every FPGA in the system. These nets are for general purpose use. Typically we call this group of nets the “YMB bus” though on older Dini Group boards, when used as the host-to-userFPGA data path, it has been referred to as “MainBus”.

System Clocking refers to the ability to run a system synchronous clock across all of the FPGA’s in the stack. It also, more generally, means that the clock networks of every board in the system must be configurable from the host.

Host Communication refers to board management functions as well as application data transfers between any FPGA in the system and a host PC (or on-board Marvell processor). On Dini Group boards this path is called “FPGABUS”, and at the board level is implemented either as “NMB bus” or “TMB bus” (See the “Dini Buses User FPGA Design Manual.pdf” for additional details about NMB and TMB busses).

II. DNNMB_CONNECT Stacking System

Virtex-7 boards (excluding DNV7F1A) support the DNNMB_CONNECT stacking system. In this system a small daughter board “DNNMB_CONNECT” plugs into the DNNMB_CONNECTOR port on each board and cables connect between them. The DINAR1 expansion ports are used for FPGA-to-FPGA interconnect.

In the DNNMB_CONNECT system, FPGA-to-FPGA Interconnect between boards is achieved with the DINAR1_CBL cable. The DINAR1_CBL connects any DINAR1 expansion header on any FPGA on any board to any other DINAR1 expansion header in the system. This allows customization of the interconnect to match the needs of the design. Each DINAR1 connection provides 150 single-ended signals or 72 LVDS pairs plus 6 single-ended.
The **YMB Interconnect** can optionally be connected between boards in the stack with a cable on the DNNMB_CONNECTOR card.

The DNNMB_CONNECTOR card provides options for **System Clocking**. System synchronous clocks can be created for up to a four-board system. Each of the global clocks can be set up for system synchronous operation across the stack, or it can use the local clocking resources independent of the board stack. Regardless of how the clocking is configured, it can all be set up from Emu through the master board connection.

**Host Communication** is chained through the stack using the DNNMB_CONNECTOR card. NMB can be chained between any number of boards. The Emu software selects one board in the stack at a time, and all commands then target that board. NMB data transfers can optionally target any FPGA on any board by directly addressing it, eliminating the need to first select the board before talking to it.

**NOTE:** If the DNV7F4A is in a stack, NMB must be run at 50Mhz on all boards in the stack. Use “Board->Set Board Info” from Emu and enter “50” for the NMB speed, leaving all other settings the same.

### III. DNBC Stacking System

Virtex-Ultrascale and newer boards support the integrated DNBC stacking system. DNBC (Dini Bank Connection) refers to the expansion headers used on these boards (see the DNBC Specification.pdf for details). There is no daughter board required for stacking in the DNBC stacking system, but each board stack must have one DNCPU board (the onboard Marvell CPU on these boards has been moved to a separate board called a DNCPU board). Only 1 CPU board is used per board stack.

In the DNBC stacking system, **FPGA-to-FPGA interconnect** between boards is achieved using DNBC cables. Each DNBC header contains 1 FPGA I/O bank of signals (51 single ended, 24 LVDS pairs). The DNBC system allows for great flexibility in the FPGA interconnect due to this single bank granularity.

The **YMB Interconnect** can optionally be connected between boards in the stack with a cable from the “YMB UP” port on one board to the “YMB DOWN” port on another board. YMB can be chained between any number of boards, but be aware of degrading performance on the bus as each board is chained in.

**System Clocking** support is provided on the DNCPU board. A single cable is run from the CLK_DISTRIBUTOR header on each board in the stack to a clock port on the DNCPU board. Up to 6 boards can share system synchronous clock resources. Any global clock on any board in the stack can be configured to (1) drive the system synchronous version of the clock, (2) use System synchronous version of the clock, or (3) use its local clock resources independent from the rest of the system. Regardless of how the clocking is configured, it can all be set up from Emu through the master board connection.

**Host Communication** (FPGABUS) is chained through the stack using cables from one board’s TMB_OUT port to the next board’s TMB_IN port. TMB can be chained through any number of boards without any degradation in performance. The Emu software selects one board in the stack at a time, and all commands then target that board. TMB data transfers can optionally target any FPGA on any board by directly addressing it, eliminating the need to first select the board before talking to it.
IV. Physical Assembly

For the DNNMB_CONNECT stacking system see the DNNMB_CONNECTOR Manual for details on making the required physical connections. For the DNBC stacking system see the DNBC Stacking System Manual. Also consult the user manual for the Dini Group boards being stacked to be sure they support the stacking system being used and for details on each of the connectors involved.

V. Accessing stacked boards from Emu

This section applies to both DNNMB_CONNECT stacking and DNBC stacking systems. The software interface is identical, and the high level software is not even aware of which system is being used.

With a fully assembled board stack and a host PC connected to the master board through any supported interface (ie Ethernet, USB, PCIe, PCI Edirect), Emu can now be used to access any board in the stack. Connect Emu to the master board using the selected interface. Then do “Board->Select Stack Level”. A list of boards in the stack will be displayed. Select a board to make it active. In the GUI build the display will immediately switch to show the selected board. In the CMD build the selected board displayed above the menu listing will switch to the newly selected board. Now all commands will affect the selected board: setting clocks, configuring FPGA’s, logic resets, FPGABUS data transfers, etc. It is as if Emu is connected directly to the stacked board.

There are several options for accessing multiple boards at one time. First, open multiple instances of the Emu program and connect one to each board in the stack. This will give you a window for each board to set up its clocks, configure FPGA’s, access its FPGABUS space, etc. Second, write custom software using the EMULIB library that opens handles to all the boards in the same application. The Emu program is designed to talk to one “global_selected_board” at a time, but this is not a limitation of the EMULIB library. Third, for FPGABUS data transfers every board in the stack can be accessed directly from the master board by adjusting the FPGABUS address. This way all of the FPGA’s in the stack can be directly accesses from a single board handle. See the next section for details on this option.

VI. Directly addressing stacked boards over FPGABUS

A host application will likely need to move data to and from the various boards in a stack quite frequently. It may be inconvenient to have to select each board before performing FPGABUS reads and writes. For this situation it is usually more efficient to keep the master board selected and have the host
application address each board directly. To do so requires an understanding of how the FPGABUS address space is constructed for a board stack.

As discussed earlier in this manual, the upper 8 bits of the 64-bit FPGABUS address is reserved for configFPGA decode of the target FPGA. For example, on a DNV7F2A Dini Group board the upper 8 bits is decoded as follows:

- **0xFE**: “Hyperpipe” [Allows management functions to be re-routed to DiniCmos]
- **0xFF**: ConfigFPGA Registers
- **0x00**: FPGA A
- **0x01**: FPGA B

...  

In a board stack, the next board in the stack begins its address space immediately following the board above it. Below is an example of a 3-board stack with a DNV7F2A, a DNV7F4A, and a DNV7F2A:

- **0xFE**: [Master] “Hyperpipe” [Allows management functions to be re-routed to DiniCmos]
- **0xFF**: [Master] ConfigFPGA Registers
- **0x00**: [Master] FPGA A
- **0x01**: [Master] FPGA B
- **0x02**: [Slave 1] “Hyperpipe”
- **0x03**: [Slave 1] ConfigFPGA Registers
- **0x04**: [Slave 1] FPGA A
- **0x05**: [Slave 1] FPGA B
- **0x06**: [Slave 1] FPGA C
- **0x07**: [Slave 1] FPGA D
- **0x08**: [Slave 2] “Hyperpipe”
- **0x09**: [Slave 2] ConfigFPGA Registers
- **0x0A**: [Slave 2] FPGA A
- **0x0B**: [Slave 2] FPGA B

So to address FPGA C on the DNV7F4A, the FPGABUS address to use is “0x06000000_00000000”. Note that when you use the “Board->Set Stacking Level” option, that FPGABUS reads and writes are automatically adjusted to use the correct address for the selected board. So FPGA A for the selected board is always at address “0x00000000_00000000”. Only use the modified FPGABUS addresses to access a stacked board when the master board is selected.

### VII. Cabled Connections Report/Verification

Data->Interconnect->Report Cabled Interconnect

Data->Interconnect->Verify Cabled Interconnect

These utilities are built on the MainRef reference design.

Report will produce a text file that lists which connectors are physically connected.

Verify will take in a text file of the same format and check that those connections in the file are present.

The files are human readable/editable.

Each line is a comma separated pair of connectors. example: P4 B0,P4 B1. P4 B0 is the connector with reference designator “P4” on board 0, which is the master board in the stack. P4 B1 is the same connector on slave board 1.

The default output file will be placed next to the Emu executable. It will be named setup_SN############.txt
11. Upgrading Software and Firmware

We are constantly working to improve the performance and reliability of the software and firmware that is provided with the Dini Group hardware products. New versions of Emu are periodically released, which sometimes include the requirement for a firmware upgrade.

I. Software Updates

Use the “Board->Software Updates->Check For Emu Updates” menu option to see if a newer release is available on the Dini Group servers (internet connection required). If a new version is available you can download the package directly from Emu by using the “Board->Software Updates->Get Latest Emu Package” option.

Alternatively, you can download the latest Emu package directly from the Dini Group website at: http://www.dinigroup.com/files/web_packs/emu.zip

The customer support package for your specific product will also be updated with the new software release. The latest customer support packages can be downloaded at: http://www.dinigroup.com/files/cust_cd
[Contact support@dinigroup.com to get username and password credentials]

We have a mailing list that sends a notice out each time a new version of Emu is released. The notice contains the release notes and the version number of the new release. To add your email address to this list, go to: http://www.dinigroup.com/mailman/listinfo/emu-update

Note that posts from list subscribers are not allowed on this list. The only emails that will ever go out on this list are Emu update announcements. Please direct all questions and comments to support@dinigroup.com.

The release package will contain full source for the current release. If you need a release that is older than what is available in the package, contact support@dinigroup.com and we will provide it for you.

II. Device Driver Updates

Updates to the device drivers are included in software releases. Check the release notes to see if updates have been made to any of the device drivers, which would then require that the updated drivers be installed on your system (and recompiled if you are using linux). This is only an issue for PCI Express users and USB users on Windows. Ethernet and USB on linux do not require a device driver.

III. Firmware Updates

The Emu software will refuse to connect to boards that have out-dated firmware. If the Emu software is updated and requires a more recent firmware version than is present on your hardware you will receive an error message describing the version conflict when you try to connect to the board. The Emu release notes indicate the minimum acceptable firmware version. You will be given the option to update the firmware or to disconnect from the board. If the firmware and software allow making a board connection but you still want to update the firmware, use the “Board->Firmware Updates” menu where
you can examine the currently installed version, install firmware images from your local machine, or installed the latest firmware images from the Dini Group servers.

Board->Firmware Updates->Update Board To Latest Firmware

Emu menu option for updating the firmware of a Dini Group board

IV. Recovering From Firmware Issues

If the Emu Software cannot update the firmware, either due to versioning issues or due to corrupt firmware on the board, there is a recovery mechanism that only requires that the bootloader SPI flash and a small portion of the NAND flash (that is separate from the root filesystem partition) are intact. This section describes the recovery procedure. Note that this procedure only updates the root filesystem partition of the NAND flash. New Dini Group boards, such as Virtex-Ultrascale based boards, have the configFPGA image stored in a dedicated SPI Flash device. After recovering the root filesystem using the procedures below, connect with the Emu Software and use the Board->Firmware Updates menu to verify that your ConfigFPGA image is up to date (and update it if necessary!).

There are two ways to recover/upgrade the root filesystem firmware on Dini Group Marvell processor based boards. The first is to do it directly over the network from the board itself, which requires the board to have an Ethernet connection that has internet access and for the board to be configured with an IP address (it uses DHCP by default, but a static IP address can be assigned if DHCP is not available). The second method is to download the upgrade package manually, place it on a USB Flash Drive, and connect the USB Flash Drive to the board for the upgrade process. Both methods are described in detail below.

DIRECT NETWORK FIRMWARE RECOVERY/UPDATE PROCEDURE

[NOTE: Boards released prior to JUNE 2013 should not use this procedure because they contain update scripts that are no longer compatible with the current firmware packages. After step 5 below, check that the file “/root/launch.sh” is present. If this file is NOT present, then you must use the USB Flash Drive procedure to update the board.]

1. Connect the board to Ethernet and remove any connected USB Flash Drives.
2. Connect an RS232 terminal to the “CPU RS232” port, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.
3. Power on the board and break into U-boot. Text will be displayed on the terminal when the board is powered on, and within a few seconds the following message will be displayed:
   
   Hit any key to stop autoboott
   
   Hit a key at this point, you will have 1 second to do so before the normal boot process will begin. If you miss it, turn the board off and try again. [You don’t have to wait for the prompt to press a key]
4. At the U-boot prompt, type the following command:
   
   run spi_boot_recoveryfs
   
   The board will now boot into the recovery filesystem.
5. When the boot is complete you will be in a linux command shell. If your network does not support DHCP, then manually configure the eth0 interface for your network. (standard Linux practices apply)

6. At the linux prompt, enter this command:
   
   sh root/recover.sh
   
   This will download the latest firmware package from Dini Group and install it. These packages are quite large and will take some time to download and uncompress. Expect up to an hour to complete this process.

7. When the upgrade is complete the following message is displayed:
   
   done with recovery procedure
   
   Type “halt” and press enter. This will flush filesystem writes and safely shut down the os. Power cycle the system to boot into the updated firmware.

   If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal. If Ethernet connectivity is a possible problem, then try the USB Flash Drive procedure instead.

**USB FLASH DRIVE FIRMWARE RECOVERY/UPDATE PROCEDURE**

NOTE: Some USB Flash Drives come formatted in a way that the Marvell linux distribution can’t read. If this seems to be affecting you please see the section in this manual titled ‘Fixing An Unreadable USB Flash Drive’.

1. Download the following files from Dini Group:
   
   http://www.dinigroup.com/~marvellfiles/rootfs.tar.bz2
   http://www.dinigroup.com/~marvellfiles/rootfs.tar.bz2.sum
   http://www.dinigroup.com/~marvellfiles/install_rootfs.sh

2. Put the files into the root of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any other USB Flash Drives or SATA Hard Drives are connected to the Dini Group board, remove them at this time.

3. Connect an RS232 terminal to the “CPU RS232” port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

4. Power on the board and break into U-boot. Text will be displayed on the terminal when the board is powered on, and within a few seconds the following message will be displayed:
   
   Hit any key to stop autoboot
   
   Hit a key at this point, you will have 1 second to do so before the normal boot process will begin. If you miss it, turn the board off and try again. [You don’t have to wait for the prompt to press a key]

5. At the U-boot prompt, type the following command:
   
   run spi_boot_recoveryfs
   
   The board will now boot into the recovery filesystem.

6. When the boot is complete you will be in a linux command shell. When you see the linux prompt, then insert the USB Flash Drive into the Dini Group Board. If your board has more than one USB port, either port may be used. Wait a few seconds for linux to enumerate the USB device.

7. Make sure all text files are formatted properly for linux by running dos2unix on them:
   
   dos2unix mnt/sda/install_rootfs.sh
**dos2unix mnt/sda/rootfs.tar.bz2.sum**

Note: Depending on the partition table of the USB Flash Drive it may mount at /mnt/sda1 instead of /mnt/sda. You can determine the mount point by looking at the output on the Marvell terminal after the USB Flash Drive was connected.

8. Enter the following command:
   `sh mnt/sda/install_rootfs.sh`
   The install_rootfs.sh script will uncompress the firmware package and install it. These packages are quite large and will take some time to install, expect about 10 minutes to complete this process.

9. When the upgrade is complete the following message is displayed:
   **done with recovery procedure**
   Type “halt” and press enter. This will flush filesystem writes and safely shut down the os.
   Power cycle the system to boot into the updated firmware.

   If any problems occur contact [support@dinigroup.com](mailto:support@dinigroup.com) and provide the exact steps that you took and the messages displayed on the terminal.

**CONFIGFPGA UPDATES**

On older Dini Group boards the ConfigFPGA image was stored in the NAND FLASH root filesystem and programmed by DiniCmos after the processor had completed booting. For these older boards, no additional steps were required to update the ConfigFPGA image; the root filesystem update would take care of it.

On Virtex-Ultrascale based boards and newer, the ConfigFPGA image is stored in a SPI FLASH device that configures the configFPGA immediately on power up. For these newer boards, the ConfigFPGA configuration SPI FLASH must be updated in addition to the NAND FLASH root filesystem. The “Board->Firmware Updates->Update Board To Latest Firmware” will take care of this automatically, but if one of the firmware recovery procedures above was used to update the root filesystem, then the configFPGA SPI FLASH must be updated separately, either by using the Emu “Board->Firmware Updates” menu or by using “DiniCmos --[path] --[stackindex]” from the onboard Marvell CPU terminal. Below is the ConfigFPGA recovery procedure which may be used to recover from a bad ConfigFPGA image loaded in the SPI Flash.

**USB FLASH DRIVE CONFIGFPGA RECOVERY/UPDATE PROCEDURE**

**NOTE:** Some USB Flash Drives come formatted in a way that the Marvell linux distribution can’t read. If this seems to be affecting you please see the section in this manual titled ‘Fixing An Unreadable USB Flash Drive’.

1. Download the ConfigFPGA bitfile for your board and configFPGA type from Dini Group: [http://www.dinigroup.com/~marvellfiles/configfpga/FPGA/bitfiles/[BOARD]/pcie_config/[FPGA]/pcie_config.bit](http://www.dinigroup.com/~marvellfiles/configfpga/FPGA/bitfiles/[BOARD]/pcie_config/[FPGA]/pcie_config.bit)
2. Put the file into the root folder of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any other USB Flash Drives or SATA Hard Drives are connected to the Dini Group board, remove them at this time.
3. Connect an RS232 terminal to the “CPU RS232” port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no
parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

4. Power on the board and break into U-boot. Text will be displayed on the terminal when the board is powered on, and within a few seconds the following message will be displayed:

**Hit any key to stop autoboot**

Hit a key at this point, you will have 1 second to do so before the normal boot process will begin. If you miss it, turn the board off and try again. [You don’t have to wait for the prompt to press a key]

5. At the U-boot prompt, type the following commands:

   ```
   setenv rw_val ro nodinicmos
   boot
   ```

6. Wait for the boot to complete and press enter to get the linux prompt

   If the board fails to reach a linux prompt, and hitting enter does not produce one, then go to the firmware recovery procedures to get a stable root filesystem installed.

7. At the linux prompt enter the command:

   ```
   DiniCmos -r -s /mnt/sda1/pcie_config.bit -c
   ```

   (sda1 is the mount point of the USB stick as mentioned before)

8. Wait for this to complete.

9. Type “halt” and press enter. This will flush filesystem writes and safely shut down the os. Power cycle the system to boot into the updated firmware.

---

**V. Linux Kernel Updates**

The latest kernel image is always available on our web server as specified below. The kernel image is very rarely updated. If a new kernel image is released it will be noted clearly in the update announcement. When a kernel update is required, follow either the network or USB procedure below:

**DIRECT NETWORK KERNEL UPDATE PROCEDURE**

1. Connect an RS232 terminal to the “CPU RS232” port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

2. Power on the board and allow it to boot normally. When it says “DiniCmos is now ready for host connections.” press enter to get a linux prompt.

3. Enter the following command:

   ```
   sh /root/launch.sh install_kernel.sh
   ```

   If launch.sh is not present on your system, then use the USB Flash Drive procedure below.

   [WARNING: If this command is interrupted while writing to the flash, the board will be unable to boot. Do not power off or otherwise interrupt this command! If the download fails it will not modify or erase the current kernel image in the flash.]

4. Enter the following command:

   ```
   reboot
   ```
Do NOT simply power off the board.
Doing a reboot guarantees that all writes are flushed out to the flash device.

USB FLASH DRIVE KERNEL UPDATE PROCEDURE

NOTE: Some USB Flash Drives come formatted in a way that the Marvell linux distribution can't read. If this seems to be affecting you please see the section in this manual titled 'Fixing An Unreadable USB Flash Drive'.

1. Download the following files from Dini Group:
   www.dinigroup.com/~marvellfiles/dini_kernel_updater
   www.dinigroup.com/~marvellfiles/uImage
   www.dinigroup.com/~marvellfiles/uImage.sum

2. Put the files into the root of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any other USB Flash Drives or SATA Hard Drives are connected to the Dini Group board, remove them at this time.

3. Connect an RS232 terminal to the “CPU RS232” port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

4. Power on the board and allow it to boot normally. When it says “DiniCmos is now ready for host connections.” press enter to get a linux prompt.

5. Insert the USB Flash Drive into the Dini Group Board. If your board has more than one USB port, either port may be used. Wait a few seconds for linux to enumerate the USB device.

6. Enter the following command:
   cd /mnt/sda
   This changes the working directory to the folder on the USB Flash Drive where the files were copied.
   Note: Depending on the partition table of the USB Flash Drive it may mount at /mnt/sda1 instead of /mnt/sda. You can determine the mount point by looking at the output on the Marvell terminal after the USB Flash Drive was connected, for the line “Hotplug(): Device Successfully Mounted At ...”

7. Enter the following command:
   ./dini_kernel_updater uImage uImage.sum
   [WARNING: If this command is interrupted while writing to the flash, the board will be unable to boot. Do not power off or otherwise interrupt this command!]
   If an error occurs that dini_kernel_updater is not executable, try chmod 777 dini_kernel_updater to force it to be executable by anyone.

8. Enter the following command:
   reboot
   Do NOT simply power off the board.
   Doing a reboot guarantees that all writes are flushed out to the flash device.

   If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal.
Kernel Developers

Users that are modifying the kernel and building their own kernel images can use the above method to load their images, but if the new image doesn’t boot properly then there will be no way to replace it. It is therefore recommended to use the method suggested by the Marvell SDK, which is to break into U-Boot and load the kernel from a TFTP server. See the section “U-Boot and Kernel Modifications” for details on the U-Boot “copy_kernel_to_flash” and “net_boot” commands.

VI. U-Boot Boot-Loader Updates

Like the linux kernel, changes are very rarely made to the U-Boot boot-loader. If a change is made it will be clearly stated in the release notes. The latest U-Boot image and source code are always available at:

http://www.dinigroup.com/~marvellfiles/u-boot-db78200_MP.bin [Binary]
http://www.dinigroup.com/~marvellfiles/u-boot-db78200_MP.bin.sum [Checksum for Binary]
http://www.dinigroup.com/~marvellfiles/uboot-[date].tar.bz2 [Source code package]

The following procedure should be followed in order to update the U-Boot boot-loader on a board.

Note that if something goes while writing the new image to the flash, then the board will be rendered un-bootable. The SF-100 programmer cable from DediProg Engineering will be required to reprogram the flash (which means the board will need to be sent back to The Dini Group to be fixed unless you want to purchase this cable).

DIRECT NETWORK U-BOOT UPDATE PROCEDURE

1. Connect an RS232 terminal to the "CPU RS232" port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

2. Power on the board and press a key to break into u-boot. Enter “printenv ethaddr” and “printenv ipaddr” and write down what they are set to. The ethaddr should match what is printed on the white sticker affixed to the RJ-45 ethernet port on the board. The ipaddr should be 192.168.2xx.xxx where the x’s are replaced by the last 5 digits of the board’s serial number. Note that this ip address is used by the board only when u-boot tries to access the network; once booted into linux this ip address is irrelevant.

3. Enter “boot” to allow the board to finish booting up normally. When it says “DiniCmos is now ready for host connections.” press enter to get a linux prompt.

4. Enter the following command:

```bash
sh /root/launch.sh install_uboot.sh
```

If launch.sh is not present on your system, then use the USB Flash Drive procedure below.

[WARNING: If this command is interrupted while writing to the flash, the board will be unable to boot. Do not power off or otherwise interrupt this command! If the download fails it will not modify or erase the current u-boot image in the flash.]

5. Enter the following command:

```bash
reboot
```

Do NOT simply power off the board. Doing a reboot guarantees that all writes are flushed out to the flash device.
6. Break into u-boot and re-set the “ethaddr” and “ipaddr” environment variables to the values you wrote down. Use “setenv ethaddr value” and “setenv ipaddr value” to do so. Then enter “saveenv” to write the new values to the flash. Power cycle the board and let it boot normally.

**USB FLASH DRIVE U-BOOT UPDATE PROCEDURE**

NOTE: Some USB Flash Drives come formatted in a way that the Marvell linux distribution can't read. If this seems to be affecting you please see the section in this manual titled 'Fixing An Unreadable USB Flash Drive'.

1. Download the following files from Dini Group:
   
   www.dinigroup.com/~marvellfiles/dini_uboot_updater
   www.dinigroup.com/~marvellfiles/u-boot-db78200_MP.bin
   www.dinigroup.com/~marvellfiles/u-boot-db78200_MP.bin.sum

2. Put the files into the root of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any other USB Flash Drives or SATA Hard Drives are connected to the Dini Group board, remove them at this time.

3. Connect an RS232 terminal to the “CPU RS232” port on the Dini Group board, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

4. Power on the board and press a key to break into u-boot. Enter “printenv ethaddr” and “printenv ipaddr” and write down what they are set to. The ethaddr should match what is printed on the white sticker affixed to the RJ-45 ethernet port on the board. The ipaddr should be 192.168.2xx.xxx where the x’s are replaced by the last 5 digits of the board’s serial number. Note that this ip address is used by the board only when u-boot tries to access the network; once booted into linux this ip address is irrelevant.

5. Enter “boot” to allow the board to finish booting up normally. When it says “DiniCmos is now ready for host connections.” press enter to get a linux prompt.

6. Insert the USB Flash Drive into the Dini Group Board. If your board has more than one USB port, either port may be used. Wait a few seconds for linux to enumerate the USB device.

7. Enter the following command:
   
   ```
cd /mnt/sda
   ```

   This changes the working directory to the folder on the USB Flash Drive where the files were copied.

   Note: Depending on the partition table of the USB Flash Drive it may mount at /mnt/sda1 instead of /mnt/sda. You can determine the mount point by looking at the output on the Marvell terminal after the USB Flash Drive was connected, for the line “Hotplug(): Device Successfully Mounted At ...”

8. Enter the following command:
   
   ```
./dini_uboot_updater u-boot-db78200_MP.bin u-boot-db78200_MP.bin.sum
   ```

   [WARNING: If this command is interrupted while writing to the flash, the board will be unable to boot. Do not power off or otherwise interrupt this command!]

   If an error occurs that dini_uboot_updater is not executable, try chmod 777 dini_uboot_updater to force it to be executable by anyone.

9. Enter the following command:
   
   ```
reboot
   ```
Do NOT simply power off the board. Doing a reboot guarantees that all writes are flushed out to the flash device.

10. Break into u-boot and re-set the “ethaddr” and “ipaddr” environment variables to the values you wrote down. Use “setenv ethaddr value” and “setenv ipaddr value” to do so. Then enter “saveenv” to write the new values to the flash. Power cycle the board and let it boot normally.

If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal.

VII. Updating the Recovery Partition

We do not recommend that anyone ever update the recovery partition of a board. This procedure is provided for completeness.

Follow exactly the steps for the U-Boot update procedures above except:

1. For Direct Network Procedure use the command:
   sh /root/launch.sh install_recoveryfs.sh
2. For USB Flash Drive Procedure download the files:
   www.dinigroup.com/~marvellfiles/rootfs.small.tar
   www.dinigroup.com/~marvellfiles/install_recoveryfs.sh
   And in step 8 run the command: sh install_recoveryfs.sh
3. The hostname in the recoveryfs filesystem must be set correctly for your board in order for the firmware update script to work properly (install_rootfs.sh). To set the hostname, do:
   sh /mnt/partition_recov/mount_recov.sh
   echo “[boardname]-[serial]” > /mnt/partition_recov/etc/hostname
   The boardname text must be exact. To discover the boardname text for your board, do:
   sh /root/install.sh
   This will print a list of valid board names. Do not run install.sh with valid parameters! This script is for factory use only, but is the controlling source for the valid boardnames.
   The serial must be a 7-digit number which is printed on a white sticker affixed to your board.
   An example of a correctly formatted hostname is “dnv7f1a-1304023”.

Note that at the factory, before a root filesystem has been installed, boards are net booted onto a “golden server” which has a root filesystem image on it, and the script “/root/install.sh” is run which installs the recoveryfs and correctly sets the hostname.

VIII. Loading a Custom ConfigFPGA Image

Some advanced features require loading a custom build of the ConfigFPGA bitfile in place of the standard build that ships with the board.

For Virtex Ultrascale and newer boards (any board that has the configFPGA bitfile stored in a SPI Flash device), see the “USB FLASH DRIVE CONFIGFPGA RECOVERY/UPDATE PROCEDURE” in the “Firmware Updates” section of this chapter. Use this procedure to load a new ConfigFPGA bitfile into the SPI Flash by using the desired bitfile instead of one from the Dini Group server.

For older boards, where the configFPGA bitfile is stored in the root filesystem, use following instructions
which describe how to copy a custom ConfigFPGA build into the correct place in the root filesystem on a Dini board.

1. Connect to the Marvell serial port (19200 bps, no parity, no flow control)
2. (Re)Boot the board and stop the board in U-Boot by pressing any key when prompted
3. Change the filesystem from read-only to read-write by executing the following command:
   > setenv rw_val r
4. Boot the board by executing the following command:
   > boot
5. If your new bitfile is stored on a USB Flashdrive, plug it in and make sure the drive mounts. If the bitfile is stored on a network drive, mount the network drive. (See chapters 7 and 8 for more details)
6. Locate the bitfile that your board is using by issuing this command at the Marvell prompt:
   > echo “display_board_info” | emu_mv -c
   The ConfigFPGA bitfile location will be included in the output. Generally, the location of the bitfile is defined as follows:
   /Dini/FPGA/bitfiles/<Board>/pcie_config/<FPGA Type>/pcie_config.bit
   Where <Board> is the name of the board and <FPGA Type> is the FPGA Type of the ConfigFPGA.
7. Overwrite the old bitfile with the new bitfile
8. Reboot the system by executing the following command:
   > reboot
   DO NOT power the card off at this point, as the filesystem is in read/write mode and writes must be flushed out to keep the filesystem coherent. Either use the “halt” command and wait for the system to go down, or use “reboot” and let the system reboot before cutting power to the card.

IX. Cloning Firmware From Existing Boards
If purchasing multiple boards it is usually desirable to have them all running the exact same firmware. If boards are purchased at different times, or returned to the factory for repair, then they will likely ship with different firmware versions installed. The “dini_firmware_cloner.sh” script was created to give customers a way to create a firmware backup from an existing board, and to install it on other boards. This includes the root filesystem (“partition_root” on the NAND Flash), the Linux kernel image, and the U-Boot image. It does NOT include “partition_user” or “partition_boardinfo” on the NAND Flash.

1. Download the following files from Dini Group:
   www.dinigroup.com/~marvellfiles/dini_firmware_cloner.sh
   www.dinigroup.com/~marvellfiles/dini_kernel_updater
   www.dinigroup.com/~marvellfiles/dini_uboot_updater

2. Put the files into the root of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any other USB Flash Drives or SATA Hard Drives are connected to the Dini Group board, remove them at this time.
3. Follow the instructions that are found at the top of the “dini_firmware_cloner.sh” script.
If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal.

IMPORTANT DETAILS:

i. Information about the board, such as the serial number and FPGA stuffing information is stored in “partition_boardinfo” on the NAND flash. This partition is not touched by the dini_firmware_cloner.sh script, so it does not need to be reprogrammed after updating a new board.

ii. The U-Boot environment variables WILL be overwritten by dini_firmware_cloner.sh to match the environment variables on the source board. As stated in the instructions, the “ethaddr” variable must be set back to the target board’s MAC address after running the script. The “ipaddr” variable should also be set to a unique address on the local network if any networking features are used from the U-Boot prompt (such as “copy_kernel_to_flash” or any other TFTP commands).

iii. “partition_user” on the NAND flash is currently not copied by dini_firmware_cloner.sh. If you are using this partition you will need to maintain your own backup separately and install it yourself on new boards.
12. FAQs, Gotchas, and Known Issues

I. Ethernet hostname registration in Windows

If your board’s hostname gets a different IP address, and windows has cached the old one, then finding the board using the hostname may fail. In this case, running “ipconfig /flushdns” will flush the windows DNS Resolver cache and force it to re-resolve the hostname to get the new IP address. This does not seem to be a problem in linux, which is better about noticing when hostnames change IP addresses.

II. Hotplugging

Connecting: You may connect a board to the system at any time on Ethernet or USB. Hotplugging PCIe is not supported (most modern operating systems do not support this cleanly). On Ethernet, it may take up to a few minutes for the board to configure itself using DHCP after it is connected, and Emu will be unable to detect the board until this completes. Typically it will come up in less than 10 seconds. The Marvell Serial Port can be monitored to diagnose problems with Ethernet configuration. If a static IP is being used, the board will be available almost immediately after the Ethernet cable is connected. On USB, the board will be detectable within a few seconds of connecting the cable.

Disconnecting: Emu will not detect when a board is suddenly disconnected from the system. On Ethernet, if the cable is unplugged or the board powered down while Emu is connected, subsequent interaction with the board will fail with error messages that may not directly indicate that the board has been removed from the system. On USB, suddenly unplugging the cable may cause unstable behavior, possibly even crashing the software. Hotplugging on PCIe is not supported and will result in unstable behavior on most modern operating systems. It is recommended to always disconnect from the board in Emu (or close the Emu software) before powering down or disconnecting the board.
III. PCIe Connectivity and Board Reset
When connecting to Dini Group boards over the PCI Express interface, enumeration occurs at the time the host system boots. A subsequent reset of the Dini Group board will erase the PCI Configuration Space registers and cause the host system to become unstable. This is typical behavior for PCI Express and is not considered a bug. Note that FPGA resets do not affect the board firmware and are expected during normal board use. Things that trigger a total board reset include pressing the “SYS_RST” button on the circuit board, issuing the “reboot” command from the Marvell linux terminal, or running the “Update Board Information” option in the Emu “Board” menu. If any of these events occur it is best to immediately reboot the host system before anything bad happens.

IV. Multiple Boards On One Host PC
The Ethernet API fully supports multiple boards one one host PC. Discovering and connecting to boards does not interfere with command processing that may be taking place simultaneously on other boards. USB and PCIe, however, have a restriction: board discovery and connection cannot take place while any other board on the system is processing commands. This is because in USB and PCIe the discovery and connection commands request board info on the same data path as all the other commands. Thus data errors will occur if any software is talking to any boards on the system during discovery and connection. The workaround for this issue is to connect to each board, one at a time, and wait until all board connections are made before starting any other commands to any boards. Once all of the board connections are made, commands to each board can run simultaneously without issue.

V. Multiple EMULIB Connections To The Same Board
It is supported to open multiple connections to the same board from different processes on the same host PC, or in the case of Ethernet from different host PC’s. However, there is only a single command data path and the EMULIB library and lower level drivers will not synchronize commands for you. Therefore running commands simultaneously on different connections to the same board is NOT supported. It is up to the software designer to implement a synchronization scheme such that only one process accesses the board at a time.

VI. Ultrascale SPI_FALL_EDGE SelectMAP loading
Boards affected: DNVU_F2PCIe

If an Ultrascale bitstream is generated using the SPI_FALL_EDGE option, that bitstream will be incapable of being loaded via SelectMAP programming, which is the programming method used to configure user FPGAs from EMU (via the configFPGA). Xilinx recommends generating a bitstream that matches the configuration method. If it is required to load an FPGA design from selectMAP and then use that design to configure the SPI, two separate bitstreams should be generated. Generating a file for SPI at a reduced configuration rate with SPI_FALL_EDGE disabled will still work on selectMAP.

VII. Ultrascale board stacking with slave boards powered down
Boards affected: DNVUF4A, DNVUF2A, DNVUF1A

If a multiboard system is connected physically with TMB connection cables and one or more slave boards are powered down while the master board is powered up, the board stack will be held in logic reset. Configuration of FPGAs will be possible, but TMB communication and user logic will be held in reset. The Logic Reset LED will be lit.