Dadio Software Development Kit

Dharma Board Documentation

<1st Draft> -

Interactive Objects, Inc.

October 2001

Author: John Locke

Table of Contents

Installation Guide

This section describes the mechanics of using the Dharma board, including:

- Building an image
- Downloading an image to the Dharma board
- Debugging an image
- Burning an image to flash
- Creating and installing a boot loader

Building the image

1. Start a bash shell and `cd` into your installation directory:

   ```
   cd /c/iobjects/dadio/sdk
   ```

2. Determine which target you want to build by looking at the available targets in `configs/targets.mk`, then build it. This generates the build tree, and creates a player image:

   ```
   make simple
   ```

3. After the build is completed, you can `cd` into the build directory to find the generated image:

   ```
   cd builds/simple
   ls -l dadio.exe
   ```

Note: Since the build tree was created in step #2, you can build directly from the build directory and save a little time from this

point on:

```
cd /c/iobjects/dadio/sdk/builds/simple
make
```

Transfering the image to the device

Now that you have created an image, you can transfer this image to the Dharma board and start debugging it, or simply transfer and execute it. this step requires that you have Dharma connected to your PC's COM port via a null modem cable.

Running the image requires the use of a gdb stub or a bootstrap environment on the board. Interactive Objects has created a combination gdb stub/bootstrap environment for Dharma using RedBoot™, a product from RedHat. The Dharma board you received should have a Redboot stub on it already. You can verify this by connecting the Dharma board, starting up a term program, and configuring it for your serial port and setting it to 115200 8N1. Next, verify that the run/program switch is in the "run" position, power on the dharma board, wait approximately 1 second, and hit the wakeup button (located on the processor mainboard). You should see some status information regarding the flash, a banner, and a prompt:

```
RedBoot>
```

RedBoot provides an interactive shell with flash management and image handling capabilities. Discussing the exact feature set is beyond the scope of this document, but is well covered at the following location:

http://sources.redhat.com/ecos/docs-latest/redboot/redboot.html

For term programs, we highly recommend TeraTerm 2.3 (Win95, 98, NT, 2k compatable), available from the following location:

http://hp.vector.co.jp/authors/VA002416/teraterm.html

If you do not see the RedBoot> prompt, you may need to burn a redboot stub to the device. This can be done as follows:

1. Power on the board.

2. Switch the board into "program" mode.
3. Hit the nPOR button on the board to induce a power-on-reset.
4. On the desktop side, open a bash shell, and cd into the sdk directory:

   ```
   cd /c/iobjects/dadio/sdk
   ```

5. On the desktop side, start the download application. If your null modem cable is plugged into COM1, the syntax is as follows:

   ```
   support/downloader.exe
   images/redboot_stub.bin
   ```

   If you are using a COM port other than 1, you need to specify the port number (X):

   ```
   support/downloader.exe -pX
   images/redboot_stub.bin
   ```

   Once this command has been issued, you should see the text:

   ```
   Waiting for the board to wakeup...
   ```

6. At this point you can hit the WAKEUP button on the Dharma processor board, and the download sequence should start.
7. When the download has completed, switch the board back into the "run" mode, start your term program as described above, then hit nPOR, wait one second, and hit WAKEUP. you should see a RedBoot> prompt at this point.

Debugging the image with gdb

Once a gdb stub has been installed on the board, it is possible to use gdb to load and debug an image. Since our image has been cross compiled, we need to use a cross debugger. Assuming you have built the target 'simple', you can perform the following steps to debug the image:

1. Open a bash shell and cd into the build directory:

   ```
   cd /c/iobjects/dadio/sdk/builds/simple
   ```

2. Prepare the Dharma(tm) board by powering it on, waiting one second, then hitting the WAKEUP button.

3. At the shell, load gdb:

   ```
   arm-elf-gdb dadio.exe
   ```

   This should bring you to the (gdb) prompt.

4. Configure gdb for the remote target:

   ```
   (gdb) set height 0
   (gdb) set remotebaud 115200
   (gdb) target remote com1
   ```

   The "height" variable controls the pager - setting it to 0 disables the pager. the "remotebaud" variable controls the baud rate of the serial target. "target remote com1" indicates that gdb should attach to a remote target on com1 - if you are using a different serial port, you should specify "comX" where X is your serial port number.

   At this point gdb will try to attach itself. if the Dharma board is not properly connected or does not have a debug stub on it, gdb on the desktop side will hang. you can CTRL-C gdb to interrupt it, then type "quit" to exit out. After the board has been properly connected and setup, repeat steps 3-4 to attach the debugger.

Note: gdb will look for and execute a startup script if one is available. this allows highly repetetive commands, such as the three listed above, to be executed automatically. to do this, simple create a file named "gdb.ini" in the same directory as your image, and put the commands in that file exactly as you would

type them at the gdb prompt.

5. Transfer the image from the host to the device:

```
(gdb) load
```

   This may take a while, depending on the size of the image. gdb will provide feedback as to which section it is currently transfering. when it is done transfering, it will return you to the (gdb) prompt.

6. Set any breakpoints you may have:

```
(gdb) break cyg_user_start
(gdb) break
main/demos/simple/src/main.cpp:84
```

   gdb will come back with the line number and file associated with the breakpoint, in addition to a breakpoint number. you can later disable or delete this breakpoint using the "disable" and "delete" commands.

7. Execute the image:

```
(gdb) continue
```

Most of our applications provide some level of feedback over the serial port. Most images will attempt to draw to the lcd, or access peripherals on the board. This should give feedback as to the status of execution. If you have breakpoints set, gdb will return you to the (gdb) prompt and indicate the breakpoint that has been hit.

Complete documentation regarding gdb syntax and features is available at:

http://www.gnu.org/manual/gdb-4.17/gdb.html

Burning the image to the device

For demos and repetitive testing, it may prove useful to burn the image out to the flash rather than transfer it over a serial connection every time. RedBoot(tm) provides flash management and can be used to assist in this as follows:

1. Start a bash shell, cd to the build directory:

   ```
   cd /c/iobjects/dadio/sdk/builds/simple
   ```

2. Convert your image to the srec format using the following syntax:

   ```
   arm-elf-objcopy -O srec dadio.exe
   dadio.srec
   ```

   This will generate the file "dadio.srec" in the current directory.

   Note: it is possible to set up a `post_link_step` in the build system to automatically perform this copy step. Refer to the build system documentation for more information.

   ---

3. Start your terminal program and wake up the board. your terminal program must support xmodem, ymodem, or zmodem for the transfer.

4. At the redboot prompt, initiate a transfer on the device side:

   ```
   RedBoot> load -m xmodem -b 0x20000 simple
   ```

   This tells redboot to start an xmodem transfer, and to place the transfered image at 0x20000 in RAM. the image is given a name, "simple", but this is not used at this point.

5. Initate a transfer on the host side. If you are using TeraTerm, this can be done by going to File-> Transfer-> XMODEM-> Send... send the previously created dadio.srec image to the device

6. When the transfer completes, RedBoot will print some gathered information about the

image:

```
Entry Point: 0x00020040, address range:
0x00020000-0x000ae1c4, length: 0x0008e1c4
```

This information will come in handy, since we are going to burn this image to the flash now.

7. Tell redboot to burn the image to the flash using the following command:

```
RedBoot> fis create -b 0x20000 -l 0x8e1c4
-e 0x20040 -r 0x20000 simple
```

"fis create" tells redboot we want to create a new image in the Flash Image System (FIS).

"-b 0x20000" indicates the base address in RAM of the image to burn, and corresponds directly to the "-b 0x20000" argument we used in the "load" operation.

"-l 0x8e1c4" tells RedBoot the length of the image - this is taken from the info line in step 6.

"-e 0x20040" tells RedBoot the entry point to the image - again, taken from the info line in step 6.

"-r 0x20000" indicates the address to load the image to. this is important since all of our images will need to be loaded to this address.

"simple" assigns a name to this image.

At this point, RedBoot will begin burning the flash with the image. when it is done, it will return to the RedBoot> prompt.

8. Verify the image was burned correctly:

```
RedBoot> fis list
```

The listing of the flash contents should have an entry for "simple".

9. Load and execute the image. It is possible to execute the image already in RAM (from step 4) without reloading it from the FIS. However, to properly verify the image, we should reset

the board and reload the image:

```
RedBoot> reset
```

The board should reset and return to the RedBoot> prompt.

```
RedBoot> fis load simple
RedBoot> go
```

This should start executing the image.

Debugging flash resident images

It is possible to mix debugging and flash-resident images, so that images are burned out to the flash via RedBoot, but can later be debugged with gdb. this is useful when an image has a condition that is difficult to trace or requires multiple runs. this requires that the image burned to flash and the image on your host are the same build. to debug this way, follow the steps listed in 2.2 up to and including step 8. at that point, proceed as follows:

1. Reset the board and load the image:

```
RedBoot> reset
RedBoot> fis load simple
```

2. close the terminal program and start gdb from your shell:

```
cd /c/iobjects/dadio/sdk/builds/simple
arm-elf-gdb dadio.exe
```

3. At the gdb prompt, execute the following commands:

```
(gdb) set height 0
(gdb) set remotebaud 115200
(gdb) target remote com1
```

At this point, gdb should attach to the remote device. the trick is to point gdb at the image to be debugged rather than the debug stub itself:

```
(gdb) set $pc=0x20040
```

This modifies the program counter to point to the entry point of the image we loaded from flash.

St this point all the standard gdb commands are available, and the image can be debugged as normal. To debug the image again, the board will have to be reset and the sequence 1-3 repeated.

Configuration and Build System

Introduction

Dadio is designed to be flexible and to maximize reuse of code. To that end, we have designed a custom build and configuration system suited to our needs. This involves organizing the source code into modules and writing custom configuration scripts for each module, in addition to parsing configuration scripts and generating appropriate build trees. Our build system uses standard `make`, and a combination of pre-existing and generated makefiles.

As a general rule, all filenames that start with an underscore ('_') are generated. Generated files live only in the build tree.

Modules: an explanation

Many sections of code in Dadio can operate independently of each other. As a result of this we have organized our directory structure and code into modules. Additionally, each module has a configuration script that allows it to specify dependencies on other modules being present in the build.

Dadio allows modules to have version numbers. The version number is present in the path for the module. This allows modules to have multiple versions present in a source tree. Module versioning is optional - it is possible to move the contents of the version number folder into its parent with no ill side effects.

Configuration system

Module configuration scripts are written in Dadio Configuration Language (DCL). Each module in Dadio has at least one DCL script that indicates what the build system should do with that module. A typical DCL file would specify a list of headers to expose and a list of source files to compile.

Building an image in Dadio requires a target. Targets reside in the `configs/` directory, and require two files: a module listing (`configs/<target>_modules`) and a makefile include (`configs/<target>.mk`). The format and details of these files will be discussed in detail later.

Targets that are available to be built are listed in the `configs/targets.mk` file, in the `TARGETS` make variable. When make is invoked at the top level, it will search the `TARGETS` variable to see if it can build the requested build target. If it is found, it will attempt to load the file `configs/<target>.mk`, then invoke the DCL script parser (`scripts/parse_dcl.pl`) on the target.

The DCL parser will step through the module list, loading each DCL file and checking its contents. Since DCL supports module dependencies, the DCL parser will verify that all module

dependencies are satisfied at this point. For each module, it will create a folder in the build directory and write out a makefile include (`_module.mk`).

If the module exports any headers, the DCL parser will create symlinks from the build tree back to the source (`player/`) tree - this allows the build tree to have the most current version of a header without performing a copy. If any source is compiled in the module, the DCL parser will create a "src" directory in the build tree.

At this point, module versioning warrants more discussion. One of the problems posed by our make system is that dependency (.d) and object (.o) files in the build tree must have the exact same path within the build tree as their corresponding source file within the source tree. However, since modules refer to headers in other modules, it is less than ideal to have version numbers in the path to our headers. As a result of this, header symlinks always reside in the base of the module path, but source files reside under the "src" directory, which resides in the version folder if one exists.

After the build tree is generated, a symlink is created to the main player makefile. The same base makefile is used for all players built under dadio.

Module lists and the target makefile include

Module lists associate a target with a list of modules and module configurations. A module list is simply a flat text file with 2 columns, delimited by whitespace. The left column indicates the path to the module that should be built in; the right column indicates the name of the configuration to use when generating a build tree for this target. If the specified configuration script is not found, the configuration "default.dcl" will be searched for. If that configuration script is not found, then the module will be excluded from the build.

It is possible to comment out a line in a module list by starting the line with the '#' character. When the list is processed, all text after the '#' character will be ignored.

The target makefile include specifies a handful of parameters and extra configuration that is independent of any given module. This file uses standard makefile syntax, and is included into other makefiles during the build process. Although many options can be configured in this file, the standard options are as follows:

Required variables:	
`ECOS_BUILD_NAME`	The name of the eCos build to link against.
`MAIN_MODULE`	The path to the module that `cyg_user_start()` is defined in. This is the entry point from the eCos bootstrap environment. This module is excluded from the build when generating tests, since the `cyg_user_start()` symbol would be multiply defined during the link step.
`TARGET_FILE_NAME`	The name of the executable image to generate from this build.
Optional variables:	
`COMPILER_FLAGS`	Additional flags to pass to the compiler.
Optional macros:	

`pre_build_step`	A macro to run prior to building sources. This macro is always executed, even if the target is up to date.
`pre_link_step`	A macro to run prior to linking the sources. This macro is only executed if a relink is required.
`post_link_step`	A macro to run after linking the sources. This macro is only executed if a relink is required.

Build system

Once the build tree has been generated, a seperate makefile is used to compile and link the player sources. This makefile includes the generated `_module.mk` files that each module in the build tree has. Once this build tree exists, it is preferable to invoke make directly from that directory, since it will avoid rerunning the configuration tool every time. The top level makefile is intended primarily to generate build trees.

A standard `_module.mk` file might look like this:

```
# builds/sdk/fs/iso/current/_module.mk
# generated by ./scripts/parse_dcl.pl on Wed
Jul 18 15:16:54 2001
# source file:
player/fs/iso/current/default.dcl

COMPILER_FLAGS += -DENABLE_ISOFS

SRC += fs/iso/current/src/cd9660_bio.c
fs/iso/current/src/cd9660_dops.c
fs/iso/current/src/cd9660_fops.c
fs/iso/current/src/cd9660_fsops.c
fs/iso/current/src/cd9660_internal.c
fs/iso/current/src/cd9660_lookup.c
fs/iso/current/src/cd9660_node.c
fs/iso/current/src/cd9660_rrip.c
fs/iso/current/src/cd9660_support.c

#
# test cases
#

iso_fs_tests :=
fs/iso/current/tests/fileio1.cpp

fs_tests += $(iso_fs_tests)

tests += $(iso_fs_tests)


## end builds/sdk/fs/iso/current/_module.mk
```

Internally, we use "current" instead of a version number for projects that are working on the tip of source control.

In this file, we can see that this module adds a parameter to the global `COMPILER_FLAGS` variable. Additionally, it adds its sources into the global `SRC` variable so they will be compiled and linked in. Nothing too fancy here.

The next part of the file deals with test programs. This module has one test program associated with it, so we have a custom

variable named '`iso_fs_tests`' associated with it. Note the next two lines however: the '`iso_fs_tests`' variable is added into 'fs_tests' and also into '`tests`'. This sets up test groupings by type, so that tests specifically for the iso FS, tests for all filesystems, or all available tests can be built.

In addition to the `_module.mk` generated for each module, a pair of top level files are generated for the build system: `_modules.h` and `_config.mk`.

`_modules.h` is a standard C header with macros defined for each module present in the system. The naming for modules is derived from their type and module name. This allows code to be conditional on whether or not a given module is available.

`_config.mk` is a makefile include that tells the player makefile which modules will be built into the system, which configuration files were used to generate those modules, and provides some other simple information to the main makefile.

When make is invoked in the build tree, it first includes the `_config.mk` file. From this file it is able to determine a list of `_module.mk` files to include and which configuration files were used to generate the build tree. If any of the configuration files are newer than the `_config.mk` file, it reruns the DCL parser to update the build tree.

Next, it includes all `_module.mk` files. From the list of compiled sources that is assembled in the `_module.mk` files, it generates a list of object files (OBJS) and a list of dependency files (DEPS). From this point on, the behavior of the makefile depends on the target:

- If a test target was specified ('`make TEST=<path_to_test>`'), the makefile includes the dependency information for that test, builds all object files in the build, and links the test.
- If a test group was specified ('`make fs_tests`'), the makefile recursively invokes itself on each test in the test grouping. When it recurses, it does not check to see if any configuration files have been updated.
- If the `config`, `clean`, or `depclean` target was specified, the makefile does not include any dependency information and will perform the

requested operation - '`config`' will force the DCL parser to run on the tree, '`clean`' will delete all object files, '`depclean`' will delete all object files and all dependency files.

- If no target is specified, the makefile includes all dependency information and builds the default target.

DCL has extensions for supporting archive generation, but this has not been tested extensively and is not currently supported. As such, the behavior of the makefile in this situation is not discussed here.

When a target is being built, make looks to make sure all the object files listed in OBJS are up to date. On a freshly generated tree, no object files or dependencies will exist. This forces make to search for sources and rebuild them. Since all source files reside in the `player/` directory, we use `VPATH` to force make to search that directory. The result is that the source is allowed to reside in the `player/` directory while object files and dependencies are placed in the build directory.

GCC has extensions that allow dependency generation and compilation in a single pass. As a result of this, when a file is being compiled a new dependency file will automatically be generated for it. GCC 2.9x has a bug that causes dependency files to be generated without path information for the target, but this bug seems to have been resolved in GCC 3.x. As such, the makefile will attempt to determine the current GCC version and compensate for this bug, at a small performance decrease.

After all object files and dependency files are up to date, the makefile will attempt to link a player with the eCos libraries. This is the final step in the build. After the build is complete, you may notice a file named "`_build_list`" in the build directory. This file lists target files that were generated during the build process. This can be useful when writing scripts to package builds, or when you are creating multiple images, as is usually the case when building tests.

Build system extensions

A brief note was made earlier about `pre_build_step`, `pre_link_step`, and `post_link_step`. These macros can optionally be defined to perform steps at certain parts of the build sequence. The syntax of these macros is best left to make documentation, such as that available for GNU make:

http://www.gnu.org/manual/make-3.79.1/html_mono/make.html#SEC55

It is important to know that all variables set in the makefile are available to these macros. As such, it may prove valuable to read through the makefile when writing build and link steps.

Introduction

Along with the source in the SDK, there is a custom configuration and build system that we use to build products based on the SDK. This system uses perl, GNU make, and the GNU toolchain to build Dadio images.

Terminology

- build
- dcl
- module
- module list
- product
- target

The base for all files in the sdk is the 'dadio' directory. For consistency purposes, references to full paths will assume you installed the SDK in "C:\dadio", which has a corresponding cygwin path of "/c/dadio".

Under the dadio directory is a product directory. In this case, the product is named 'sdk'. Internally, the SDK is treated similarly to any other product; it has its own top level directory in the dadio directory, and a complete directory structure below it.

In the product directory you will find the following items:

Makefile

The top level product Makefile. This allows you

to create and remove build trees for a specified target.

configs/

A directory with files for each target, and a makefile include (targets.mk) which specifies the available targets.

images/

Pre-built utility and demo images that you can burn onto Dharma and experiment with, including documentation describing how to use them.

docs/

Documentation for the SDK.

ecos/

A directory with builds of various kernel configurations.

player/

All the code and libraries shipped with the SDK.

scripts/

Support utilities and scripts used by the build and configuration system. You shouldn't ever need to modify the contents of this directory.

support/

Various images and utilities for downloading images to and configuring the board.

Configuration File

Introduction

Configuring the Dadio player system takes a significant amount of work. Using flexible modules allows the maximal reuse of code, but hardware drivers and other instance bound portions of code need to have maximal configurability. Relying on the preprocessor to provide conditional code for each target quickly becomes unmanageable, and making multiple copies of varies files for individual targets reduces code reusability. As such, it is important to have a pre-build time configuration step to target the player towards a specific platform.

Overview

In order to minimize code changes for new hardware, hardware dependancies that can be reduced to preprocessor macros and compiler flags that are specified in a .cfg file in the module base directory. Inclusion of a specific module is then reduced to picking the correct configuration of that module, or creating a new configuration for a new target. Existing systems for accomplishing this are relatively complex and require additional programs to support them, since they tend to focus more on the dependancy tree creation that the issue of module configurability. As a result, the best plan is to create a simple scripted system for tree configuration.

Dadio Configuration Language (DCL)

Configuration files are written in DCL. DCL is a very simple language. commented lines begin with the # character. The text for any directive must fit on a single line. Blank lines are ignored.

There are a few main keywords in DCL:

`name` *<module_name>*	Specifies a name for this module. Other modules can then explicitly use this module as a dependency using the name/type pair.
`type` *<module_type>*	Specifies a type for this module. The currently allowed types are: `storage` Storage (block) device `net` Network device `dev` Generic device `input` Input wrapper `output` Output wrapper `filter` Filter/codec wrapper `fs` Filesystem `playlist` Playlist `other` Unspecified type
`requires` *<packagename>*	Specifies that this module needs the package <packagename> to be built in order to function. <packagename> can either be a specific package, or a wildcard. An example is that a fat module (of type fs) would require a module of type any_storage to be built in. An iso9660 module would require a

	module of type ata_storage be built (since the ata module includes the atapi support for CD drives).
`build_flags <build options>`	A string to be added to the build line for gcc and g++.
`link_flags <link options>`	A string to be added to the final link step.
`tests <file list>`	Specifies test targets that can be built for this module.
`export <file list>`	Specifies the headers in this module that should be exported to the build tree. Headers not listed in the export list will not be accessible to other modules.
`compile <file list>`	Specifies the source files in this module that must be compiled for the module to function.
`link <file list>`	Specifies objects or libraries to be linked at the final link step.
`arch <object list>`	Specifies a list of object files to archive into a library.
`header <headername> <start | end>`	Specifies the start (or end) of a block of text to be generated and deposited into the header file specified by <headername>. This header exists in the build tree under the directory for this module. An example of this would be as follows:

```
header hw_cfg.h start
#include
<cyg/hal/hal_edb7xxx.h>
#define HW_TARGET_NAME
"dharma"
#define HW_GPIO PDDR
#define HW_RESET 0x04
header hw_cfg.h end
```

This would generate the header hw_cfg.h in the build tree, with the following contents:

```
// generated file, do not edit
!!
#ifndef __HW_CFG_H__
#define __HW_CFG_H__

#include
<cyg/hal/hal_edb7xxx.h>
#define HW_TARGET_NAME
"dharma"
#define HW_GPIO PDDR
#define HW_RESET 0x04

#endif // __HW_CFG_H__
```

The most basic DCL file must have a name and a type specified, but does not need to use any other directives. The simplest sensible DCL file would have at least one file that it compiled, or at least one header that it exported.

Hardware Information

Dharma Board Hardware

The documentation for the Dharma Board is all shipped in PDF format. If you need a viewer for this, install the free Acrobat Reader software included on this CD-ROM.

Here are some generic overview documents:

SDK Product Bulletin

This is the two-page marketing document for this SDK.

Layout Application Notes

These application notes are to help you design the layout of the DAC, ADC, and Codec components.

7312 Chip Documentation

The Dharma Board is designed to work with three chips: the ep7312, the ep9312, and the xxxx.

7312 Product Bulletin

This is a four-page marketing description of the EP 7312 chip.

7312 Data Sheet

A 38-page data sheet summarizing the features and configuration of the 7312 chip.

User Manual

The complete user manual for the 7312 chip.

Errata

Changes and notes found since the publication of the above documents.

Analog to Digital Converter Chip

The Dharma Board comes with the CS53L32A Analog-to-Digital Converter.

CS53L32A Data Sheet

38-page document detailing the ADC.

ADC Evaluation Board

Data sheet describing how to test the ADC.

Digital to Analog Converter Chip

The Digital to Analog Converter (DAC) is the output device that changes the raw digital signal to an analog signal and amplifies it to the correct level. The Dharma board is designed to use the CS43L43 DAC.

CS43L43 Data Sheet

38-page document detailing the DAC.

Errata

An omission from the above document.

CS8900a Ethernet Chip

The Dharma board is designed to work with the CS8900a Ethernet chip.

Features

A marketing summary of the features of the CS8900a chip.

Product Data Sheet

Complete reference for the CS8900a product.

Errata

Changes to the above document.

Programmable Logic Device

The Dharma Board is designed to use the Cypress Ultra37000™ CPLD family for programmable chips for your application.

CPLD documentation

This is the user manual for the Cypress Ultra37000 CPLD.

USB Documentation

The Dharma Board is designed to use the Philips PDIUSBD12 Universal Serial Bus (USB) to connect to miscellaneous devices.

Data Sheet

Hardware specification for the PDIUSB12 bus.

Programming Guide

Programming guide for interacting with devices on the USB.

Hardware Release Notes

text

Bill of Materials

Download the Bill of Materials in PDF format here.

I/O locations

This PDF lists the I/O locations for each supported processor.

Dharma Jumper Header Descriptions

This PDF describes the jumpers exposed on the Dharma board.