**Anacaz**

**Integrated Diagnostics System (IDS)**

**Functional Specification**

***Anacaz Networks, Inc.***

San Jose, California

Revision History

| Version | Description | Updated By | Date |
| --- | --- | --- | --- |
| 0.1 | Initial Draft | Ryn Ryndyk | 11/19/2010 |
| 0.2 | Revised Draft | Ryn Ryndyk | 12/07/2010 |
|  |  |  |  |

Table of Contents

[1. Overview 3](#_Toc279498955)

[1.1 Document Purpose 3](#_Toc279498956)

[1.2 What is IDS? 3](#_Toc279498957)

[1.3 U-boot Connection 3](#_Toc279498958)

[**1.4** **IDS Operations** 3](#_Toc279498959)

[**1.4.1** **Download TLS Files** 3](#_Toc279498960)

[**1.4.2** **Build Master List Menu** 3](#_Toc279498961)

[**1.4.3** **Test Execution (Menu Driven)** 8](#_Toc279498962)

[2. U-boot Directory Structures 8](#_Toc279498963)

[3. Core IDS 9](#_Toc279498964)

[**3.1** **Source Files** 9](#_Toc279498965)

[**3.2** **Makefile** 10](#_Toc279498966)

[**4.** **IDS Commands** 11](#_Toc279498967)

[**4.1** **help** 11](#_Toc279498968)

[**4.2** **poke [-bwlr] address data [count]** 11](#_Toc279498969)

[**4.3** **peek [-bdhloqw] address [count]** 12](#_Toc279498970)

[**4.4** **ram-addr-hi address size** 14](#_Toc279498971)

[**4.5** **ram-addr-lo address size** 14](#_Toc279498972)

[**4.6** **ram-alias address size** 15](#_Toc279498973)

[**4.7** **ram-cache address size** 16](#_Toc279498974)

[**4.8** **ram-refresh address size** 16](#_Toc279498975)

[**4.9** **ram-readback address size** 16](#_Toc279498976)

[**4.10** **reg-rw access reg1[: mask1] reg2[:mask2] reg3[:mask3] …** 17](#_Toc279498977)

[**4.11** **reg-ro access reg1 reg2 reg3 …** 17](#_Toc279498978)

[**4.12** **reg-wo access reg1 reg2 reg3 …** 18](#_Toc279498979)

[**4.13** **show - update menu hw register items** 18](#_Toc279498980)

[**5.** **Tools - /home/<username>/uboot/tools/ids/** 18](#_Toc279498981)

[**5.1** **mktest** 18](#_Toc279498982)

[**5.2** **mklist** 19](#_Toc279498983)

[**5.3** **showlist** 19](#_Toc279498984)

[**6.** **Builds** 19](#_Toc279498985)

[**6.1** **U-boot Image Build** 19](#_Toc279498986)

[**6.1.1** **IDS Core** 19](#_Toc279498987)

[**6.1.2** **TCM Board-specific** 20](#_Toc279498988)

[**6.2** **Standalone Test Image Builds** 20](#_Toc279498989)

[**6.2.1** **TCM Standalone Tests** 20](#_Toc279498990)

[**7.** **Config Files** 21](#_Toc279498991)

[**7.1** **../uboot/include/config.mk** 21](#_Toc279498992)

[**7.2** **Board specific config files** 21](#_Toc279498993)

[**8.** **TCM Menus** 21](#_Toc279498994)

[**8.1** **Core menus – Compile time** 22](#_Toc279498995)

[**8.1.1** **Main Menu** 22](#_Toc279498996)

[**8.1.2** **Mode Menu** 22](#_Toc279498997)

[**8.1.3** **Execution Menu** 23](#_Toc279498998)

[**8.1.4** **RAM Carve Menu** 23](#_Toc279498999)

[**8.1.5** **TFTP File Menu** 24](#_Toc279499000)

[**8.1.6** **Xmodem Menu** 24](#_Toc279499001)

[**8.2** **Core menus –Run time** 24](#_Toc279499002)

[**8.2.1** **Log Menu** 24](#_Toc279499003)

[**8.2.2** **List Menu – Master List Menu** 25](#_Toc279499004)

[**8.2.3** **Debug List Menu** 26](#_Toc279499005)

[**8.3** **Test menus – Compile time** 26](#_Toc279499006)

[**8.3.1** **File Manager Menu** 27](#_Toc279499007)

[**8.3.2** **File Manager Config Menu** 27](#_Toc279499008)

[**8.3.3** **File Manager EDVT Menu** 27](#_Toc279499009)

[**8.3.4** **File Manager a6addr Menu** 28](#_Toc279499010)

[**8.3.5** **File Manager FPGA Menu** 28](#_Toc279499011)

[**8.3.6** **File Manager CHAN Menu** 29](#_Toc279499012)

[**8.3.7** **File Manager a6addr Debug Menu** 30](#_Toc279499013)

[**9.** **Download Process** 30](#_Toc279499014)

[**9.1** **TFTP File Menu** 30](#_Toc279499015)

[**9.2** **Downloading TLS Files** 31](#_Toc279499016)

[**9.2.1** **TLS File Memory Allocation Storage** 31](#_Toc279499017)

[**9.2.2** **Auto Menu Build** 31](#_Toc279499018)

[**10.** **Test Execution Flow** 32](#_Toc279499019)

[**10.1** **test\_go\_appl(menu\_t \*p)** 33](#_Toc279499020)

[**10.2** **do\_test\_cycles\_appl(int cycles, int (\*test\_appl)(menu\_t \*p), menu\_t \*p)** 33](#_Toc279499021)

[**10.3** **do\_test\_go\_appl(menu\_t \*p)** 33](#_Toc279499022)

[**10.4** **do\_test\_appl(menu\_t \*p, char \*\*action)** 34](#_Toc279499023)

[**10.5** **Pass/Fail Policy Actions** 35](#_Toc279499024)

[**10.5.1** **Edit Pass/Fail Policy** 35](#_Toc279499025)

[**10.5.2** **<selection:2-16>** 36](#_Toc279499026)

[**10.5.3** **next** 36](#_Toc279499027)

[**10.5.4** **stop, abort, halt** 36](#_Toc279499028)

[**10.5.5** **ignore** 36](#_Toc279499029)

[**10.5.6** **skip** 37](#_Toc279499030)

[**10.5.7** **retry** 37](#_Toc279499031)

[**10.5.8** **pause** 37](#_Toc279499032)

[**10.5.9** **back** 37](#_Toc279499033)

[**10.5.10** **spin** 37](#_Toc279499034)

[**10.5.11** **wait** 37](#_Toc279499035)

[**10.5.12** **sleep** 37](#_Toc279499036)

[**10.5.13** **loop** 38](#_Toc279499037)

[**11.** **U-boot Jump Vector Table** 38](#_Toc279499038)

[**11.1** **load\_u\_boot\_vectors()** 38](#_Toc279499039)

[**11.2** **ids\_jmpvec.h** 39](#_Toc279499040)

[**11.3** **ids\_jmpvec.c** 39](#_Toc279499041)

# Overview

## Document Purpose

The purpose of this document is to provide a thorough description of IDS as it currently exists. That includes all standard operating modes and processes as well as features targeting specialized needs such as bring up. If, after review of this document, it is deemed that some features are no longer needed or that other features must be added, then this document can be revised accordingly.

## What is IDS?

The Integrated Diagnostics System (IDS) is a framework for executing diagnostic tests in a controlled environment. IDS has a menu-driven user interface where the menus provide the means to download standalone diagnostic tests, execute them and capture test results and status. IDS also provides a mode called “**AUTORUN**” to execute a pre-defined set of diagnostics for regression testing and manufacturing burn in.

## U-boot Connection

IDS has been incorporated into the u-boot command set. The core IDS source files, as well as all test source files, have been integrated into the u-boot code set.

During u-boot boot up, entering any key puts the user into the u-boot CLI mode. Typing “ids” and a return brings up the IDS main menu. From that point, the user can traverse the menus to download tests, execute them and capture results.

* 1. **IDS Operations**
     1. **Download TLS Files**

Diagnostic tests that have been compiled during a test-specific build (separate from the uboot build) as standalone tests are copied by the user to the tftp directory on a tftp server. IDS provides the means to download tests under user control from the server to designated memory on the unit under test (UIT).

* + 1. **Build Master List Menu**

Once download is complete, IDS parses the headers of the downloaded images. A TLS header is a wrapper for up to 15 tests with an IDS header for each test. During test image build, each executable is processed by the “mktest” tool, as called out in the local makefile, into an .ids file which has an IDS header that includes default arguments for the test as well as the executable image. Then, up to 15 IDS files can be collected into one TLS file by the “mklist” tool with an extension of .tls. The TLS header contains information about all IDS files contained in the TLS file.

Using the information contained in the various TLS and IDS headers, IDS builds up a Master List Menu through which the user can execute the tests that have been downloaded. The maximum number of TLS entries in this menu is 15.

See section 8 for an explanation for the 15 entry limit.

* + 1. **Test Execution (Menu Driven)**

After the Master List Menu is finished building, the user can initiate execution of the master test list by navigating through the menus to the Master List Menu and selecting “1) RUN”. This will cause IDS to traverse all the TLS entries at this level in the order in which they occur in this master list and execute all the tests specified by each TLS entry.

If a test fails, what IDS does next is defined by the pass/fail policy stipulated in the Makefile when the test is built. If the Makefile does not explicitly call out a pass or fail policy, then the default action is taken. The default action for PASS is to run the next test in the list. The default action for FAIL is to stop (end this test run, show the FAIL message and wait for any user input to return to the Execution Menu).

One final note: if **AUTORUN** is selected to run the pre-set list of TLS files, the **STOP** on failure feature is disabled such that when a test fails, the log shows the failure and the test run continues with the next test in the list.

Rather than running all tests, the user can traverse the master list menu to a specific TLS entry and execute all the tests it contains. And, finally, the user can traverse the TLS menu to a specific test and execute just it.

The code flow through the controlling functions is described elsewhere in this document.

# U-boot Directory Structures

In the Anacaz code repository, there is a single u-boot directory with board-specific differentiation for builds provided by board-specific configuration files and board-specific directories for non-u-boot builds, i.e., standalone IDS tests.

For development purposes, it could be less confusing to have separate complete u-boot directories on a per board basis. An example of a such a board-specific development directory, using the TCM board, is:

**/home/<username>/tcm/uboot/**

In each board-specific directory path, the board-specific u-boot build is performed in the **../<board>/uboot/** directory. The primary build files for either board’s build are:

**../uboot/Makefile**

**../uboot/config.mk**

**../uboot/include/configs/anacaz\_tcm.h** – contains TCM-specific declarations

In each board-specific directory structure path are IDS core source files and standalone diagnostic test source files. Conditional compiler flags are defined in the respective header file, anacaz\_tcm.h, for use in compiling IDS core source files into a board-specific version of u-boot. Test source files are in board-specific directories and are discussed elsewhere in this document.

For purposes of discussion in the rest of this document, the TCM directory will be used as an example.

# Core IDS

The path to core IDS directories is:

/**home/<username>/uboot/lib\_ids/**

All the core IDS source files are located in this directory.

There are also 2 additional sub-directories at this level:

**/home/<username>/uboot/lib\_ids/built-ins/**

**/home/<username>/uboot/lib\_ids/hw/**

Most source files are included in the u-boot build while a few are not. The following section identifies which source files are included and which are not.

* 1. **Source Files**

The source files that are included in the u-boot build are:

builder.c - builds the menus

exec.c - controls test execution

fileio.c - support for I/O: tftp, log file, etc.

cmd.c - IDS based commands: peek, poke, etc.

main.c - IDS start up code, u-boot vector table, etc.

menu.c - menu processing support; includes user interface

menutab.c - IDS control menus, TLS test menus, menu application functions

flash.c - support for flash devices, not fully implemented

memtools.c - memory tools: peek and poke

patgen.c - pattern generator support

pgen.c - packet generator support

progress\_bar.c - progress bar support; incomplete

built-ins/ids\_nand.c - support for NAND flash memory

built-ins/ids\_tcm.c - A6 CODEC test support

built-ins/ids\_ram.c - RAM test support

The remaining source files in the ../lib\_ids/built-ins/ directory and all the files in **../lib\_ids/hw/** are not included in the build.

The criteria for including files in the u-boot build falls into 2 categories: core IDS files from the **.. >/uboot/lib\_ids/** directory are automatically in the build and any built-in tests the developer deems as required. It is possible, and perhaps even desirable, to limit u-boot build files to core IDS, only and to move the built-in tests to the standalone category.

* 1. **Makefile**

The core IDS build is part of the full u-boot build. IDS is defined in u-boot as a command that is issued at the u-boot CLI. The u-boot Makefile, which includes commands to build IDS core, is located in: /home/<username>/tcm/uboot.

Instructions for executing the u-boot build are in Anacaz Wiki at:

<https://wiki/mywiki/MP_notes/TCM_notes/TCM_uboot>

**make tcm\_config:**

After traversing to the “uboot” directory, the user first executes the command “make tcm\_config”. The makefile executes a script named “mkconfig” to create header files and links to configure u-boot for the TCM board. The arguments passed into this script includes the string “anacaz\_tcm” which the script uses to add the file “anacaz\_tcm.h” to the build. This file contains TCM board-specific macro and compiler definitions. The actual file is located at: **../uboot/include/configs/**

**make:**

The user then issues the command “make”. The Makefile is executed and builds a board-specific uboot image that includes the IDS core code.

1. **IDS Commands**

IDS commands were originally provided for use during bringup. They allowed the user to run tests or set up conditions for tests that helped in identifying hardware issues or verifying hardware performance.

For example, one of the memory tests could be performed several times over a specified memory region and have the last test execution timed to get a baseline for how long the test takes to complete running from cache. Next, the cache read utility could be executed over a different equally sized memory region to flush the existing contents of L2 cache and fill it with the new memory region’s content. Finally, the original memory test could be re-run over the original memory region and timed. If the time for the second run exceeds the time for the first run, that would indicate that L2 cache is performing correctly. The second run should be longer because of all the cache misses.

The commands are described here in the interest of having IDS fully documented. Whether or not it’s desirable to leave the commands in IDS is a discussion for a different forum.

* 1. **help**

The “help” command causes IDS to display all the commands it provides along with the arguments each command requires.

* 1. **poke [-bwlr] address data [count]**

Function location: **../uboot/lib\_ids/memtools.c**

The **poke** command provides the means to write data to locations in memory. It takes both optional and required arguments.

The optional argument **[bwlr]** represents the data type being written: **b** = byte (8 bits, DEFAULT), **w** = word (16 bits), **l** = long (32 bits) and **r** = follow **poke** with **peek** using the same arguments. In the absence of a data type, both **poke** and **peek** default to bytewise operations. The other optional argument **count** provides the number of consecutive memory locations to **poke** with data and defaults to 1.

The required argument **address** is the memory location to start poking and **data** is the actual data to **poke** into memory.

There is a tightly coupled relationship between the data type, the address and the count arguments. A bytewise **poke** will write the byte **data** starting at the **address** specified for **count** consecutive times. For example:

poke 0x00400000 0x41 16

will put the data value 0x41 (‘A’) into the 16 consecutive byte locations starting at address 0x004000 which would look like (all values are hexadecimal):

00400000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

If the data type selected is **w** (16 bits), the command:

poke -w 0x00400000 0x4142 8

will put the 16-bit value 0x4142 (“AB”) into the 8 2-byte locations starting at address 0x00400000 which would look like (all values are hexadecimal):

00400000: 4142 4142 4142 4142 4142 4142 4142 4142

The **–r** option of the data type argument can be used in conjunction with the other data type options. Using it causes the **poke** operation to be immediately followed by a **peek** operation to confirm that data was written correctly.

For instance, the first example above could be written as:

poke -r 0x0040000 0x41 16

The **poke** operation would behave identically to the first example. The **–r** option, however, would invoke the **peek** operation after the **poke** operation concluded using the **poke** operation’s data type, starting address and count to display the data just written to memory. For the other data types, they would be explicitly called out in combination withthe **–r** option:

poke -wr 0x00400000 0x4142 8 - wordwise operations, **peek** follows

poke -lr 0x00400000 0x41424344 4 - longwordwise operations, **peek** follows

poke -l 0x00400000 0x30313233 4 - longwordwise operations, no **peek**

* 1. **peek [-bdhloqw] address [count]**

Function location: **../uboot/lib\_ids/memtools.c**

The **peek** command provides the means to read data from locations in memory. It takes both optional and required arguments.

The optional argument **[-bdhloqw]** flags the data type being read, in what radix to display the data and a run quiet flag to disable data display. The argument options **[bwl]** represent the data type being read: **b** = byte (8 bits, DEFAULT), **w** = word (16 bits) and **l** = long (32 bits. The argument options **[ohd]** represent the output format for the data: **o** = octal, **h** = hexadecimal and **d** = decimal. The **[q]** option dictates a run quiet mode, i.e., disable display of data read from memory. In the absence of any data type, **peek** defaults to: bytewise data width, count of 1 and hexadecimal output.

The other optional argument **count** provides the number of consecutive memory locations to **peek** for data and defaults to 1.

The required argument **address** is the memory location to start peeking.

There is a tightly coupled relationship between the data width, the output format, the address and the count arguments. A bytewise **peek** will read the byte **data** starting at the **address** specified for **count** consecutive times and display it in the specified output format. The size of the output data is tied to the specified data width. For example:

peek 0x00400000 16

read the 16 bytewise locations starting at 0x00400000 and display them as 16 individual hexadecimal byte values:

0x00400000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

If the data type selected is **w** (16 bits), the command:

peek -w 0x00400000 16

reads the 16 wordwise (16 bits) locations starting at 0x00400000 and displays them as 2 rows of 16 bit values:

0x00400000: 4141 4141 4141 4141 4141 4141 4141 4141

0x00400010: 4141 4141 4141 4141 4141 4141 4141 4141

An example of a longwordwise read and an octal display follows of locations in memory that were all written with 0x41:

peek –lo 0x00400000 16

00020000000: 10120240501 10120240501 10120240501 10120240501

00020000020: 10120240501 10120240501 10120240501 10120240501

00020000040: 10120240501 10120240501 10120240501 10120240501

00020000060: 10120240501 10120240501 10120240501 10120240501

* 1. **ram-addr-hi address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-addr-hi.c**

The **ram-addr-hi** command executes the address stuck high test. It takes 2 required arguments: the **address** of the memory location for the start of the test and the **size** (in bytes) of the test memory region.

The test uses the first longwordwise location of the **address** as a base location. The locations to which the foreground pattern is written are calculated by shifting a 1 bit left through the base location address starting with bit 3. So, assuming an **address** of 0x00400000, the base location is 0x00400000. The address sequence for the foreground pattern then follows as: 0x00400004, 0x00400008, 0x004000010, 0x00400020, etc.

First, the test writes the background pattern to the base location. Then the test steps through the foreground locations. For each location, it writes a foreground pattern and immediately reads the base location. If the pattern read is not the background pattern, the test has failed and immediately returns with a failed return code. FAIL is either the foreground pattern is found or the background pattern is not found. PASS indicates the background pattern is found.

* 1. **ram-addr-lo address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-addr-lo.c**

The **ram-addr-lo** command executes the address stuck low test. It takes 2 required arguments: the **address** of the memory location for the start of the test and the **size** (in bytes) of the test memory region.

The test uses the first longword location of the **address** as a base location. The locations to which the foreground pattern is written are calculated by shifting ‘1’ left through the address starting with bit 3. So, assuming an **address** of 0x00400000, the base location is 0x00400000. The address sequence for the background pattern then follows as: 0x00400004, 0x00400008, 0x004000010, 0x00400020, etc.

The test first writes the background pattern to all locations. The background pattern write ends when the shifted address reaches the limit set by **size**. The base location is then written with the foreground pattern. Finally, all background patterns are read to confirm the background pattern has not changed. FAIL is either the foreground pattern is found in a background location or the background pattern is not found. PASS indicates the background pattern is found at all background locations.

* 1. **ram-alias address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-alias.c**

The **ram-alias** command executes the memory aliasing test. It takes 2 required arguments: the **address** of the memory location for the start of the test and the **size** (in bytes) of the test memory region.

The test algorithm is:

* fill memory using longword writes in ascending order from **address** to the end of the memory region specified by **size**. At each longword location write the location address
* display memory with longword reads for 400 locations starting at (**address** - 64). Should be garbage for the first 64 bytes followed by the address values from the previoius write operation.
* verify that the original memory locations written have not changed by performing longword reads in ascending order starting at **address** to the end of the memory region originally written.
* display memory starting at the next memory region (**address + size**) – 64 to verify that no aliasing took place…a visual exercise here.
* fill the specified memory region with 0
* display the specified memory region for 400 locations to show that memory is 0
* write the specified region in memory in descending order using longword writes and starting at the end of the memory region. At each longword location write the location address.
* display memory in ascending order with longword reads for 400 locations starting at address **- 64**. Should be garbage for the first 64 bytes followed by address values from the previous write operation.
* verify that the original memory locations written have not changed by performing longword reads in ascending order starting at **address** to the end of the memory region originally written.
* display memory starting at the next memory region (**address + size**) – 64 to verify that no aliasing took place…a visual exercise here.

Any verification mismatch is an immediate FAIL.

* 1. **ram-cache address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-alias.c**

The **ram-cache** command executes the cache fill operation. It takes 2 required arguments: the **address** of the memory location for the start of the cache fill and the **size** (in bytes) of the cache fill memory region.

This is not a test. It just takes the parameters provided by the user and reads the memory region specified in character-sized operations into cache.

* 1. **ram-refresh address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-refresh.c**

The **ram-refresh** command executes the memory refresh test. It takes 2 required arguments: the **address** of the memory location for the start of test and the **size** (in bytes) of the memory region.

The test is a CHAR (8 bit) access test of the range of memory defined by the arguments. The 8 bit hamming pattern used is: 0x00, 0xFF, 0xF0, 0xCC, 0xAA and 0x55. Each pattern is written to the entire memory range followed by a read and verify operation across the memory region. Any mismatch during the verify stage results in the test ending with a FAIL status.

* 1. **ram-readback address size**

Function location: **../ uboot/board/anacaz/ids/ram/ram-readback.c**

The **ram-readback** command executes the memory readback test. It takes 2 required arguments: the **address** of the memory location for the start of test and the **size** (in bytes) of the memory region.

The test is a CHAR (8 bit) access test of the range of memory defined by the arguments. The 8 bit hamming pattern used is: 0x00, 0xFF, 0xF0, 0xCC, 0xAA and 0x55. Each pattern is written to a location in memory and then immediately read back and verified. This sequence is repeated over the entire memory region. Any mismatch during the verify stage results in the test ending with a FAIL status.

* 1. **reg-rw access reg1[: mask1] reg2[:mask2] reg3[:mask3] …**

Function location: ../ uboot/board/anacaz/ids/lib/reg-rw.c

The **reg-rw** command executes the device register read/write test. It takes 2 required arguments: **access** is the register size in bits (“char”, “short”, “long” or “longlong”) and **reg1** is the register’s address. Some addresses may be memory mapped while others represent values needed to indirectly access registers in other devices such as the FPGA. An optional extension to the register argument is **:mask1** which represents the writeable bits of the register specified in **reg1**. Multiple register addresses, and their optional mask extensions, can be included up to a total of 15 register-mask sets.

Note that the full format for a single register argument is: **<reg>:<mask>**

The test uses the **access** argument to identify the register size. If the optional register extension **:mask<n>** is not provided, then the mask for this register test defaults to all bits as writeable, i.e., “char” size mask is 0xFF, “short” size mask is 0xFFFF, etc. The test is performed on the first register with a 1 bit, moderated by the mask, being walked through the register from least significant bit to the most significant bit. Each write to the register is followed by a read. If a mismatch occurs, then the test ends with a FAIL status.

* 1. **reg-ro** **access reg1 reg2 reg3 …**

Function location: **../ uboot/board/anacaz/ids/lib/reg-ro.c**

The **reg-ro** test executes the read test across all specified registers using **access** to identify the register size. Each register entry, i.e., **reg1**, **reg2**, etc., is the address for that register. Some addresses may be memory mapped while others represent values needed to indirectly access registers in other devices such as the FPGA.

For each register, the test reads the contents of the register. It then performs as many reads of the same register as there are bits in the register size (8, 16, 32 or 64). If any test read doesn’t match the original read, then the test ends with a FAIL status. The test continues testing registers until it exhausts the registers specified in the command line.

* 1. **reg-wo access reg1 reg2 reg3 …**

Function location: **../ uboot/board/anacaz/ids/lib/reg-wo.c**

The **reg-wo** test executes the function across all specified registers using **access** to identify the register size. Each register entry, i.e., **reg1**, **reg2**, etc., is the address for that register. Some addresses may be memory mapped while others represent values needed to indirectly access registers in other devices such as the FPGA.

For each register, this function walks a 1 bit through the register from least significant bit to most significant bit. It does no checking after any write. It always returns PASS unless an error occurred in the actual command entry.

* 1. **show - update menu hw register items**

Function location: **../uboot/lib\_ids/menu.c**

This command was originally intended to assist in the bring up of the FPGA. It provided a mechanism by which indirect reads of the specified FPGA registers could be presented to the user in the menu.

This command takes its argument, a menu pointer, from the global variable **this\_menu** which gets set by both menu processing routines **menu\_input()** and **list\_input()**. Both functions are in this file. The pointer is set when either a menu or a list is processed. This command only executes if the menu in question is designed to access registers. Otherwise, the command does nothing.

Using the menu pointer argument, the command walks the parameters in the menu and updates all specified register parameters’ local data storage by reading the target FPGA register contents and storing them in the local data location.

1. **Tools - /home/<username>/uboot/tools/ids/**

The tools described in this section are standalone programs used by Makefile at compile time to create test (**mktest**) files and test list (**mklist**) files and to display the contents of the list files (**showlist**). The **showhex** and **bin2ascii** are not used. All tools are executable from the unix/linux command line as well as within a Makefile.

* 1. **mktest**

This tool takes the arguments called out in the Makefile and converts them into an .ids file that includes an IDS test header, parameters as ascii text and executable image.

There 2 types of IDS files that can be created. First, there are test files that represent actual tests to be performed on the target board. Then there are files that perform a service such as a TFTP download of an FPGA image to target board memory and other service files that take that downloaded FPGA image and load it to the FPGA device.

* 1. **mklist**

This tool takes the IDS file arguments, along with pass/fail actions, wraps them with a TLS list file header and saves it all to a .tls file. The IDS file order specified in the Makefile defines the IDS test file order in the TLS file.

* 1. **showlist**

This tool takes a TLS file as an argument and displays the contents, minus the executable image, in a readable format on the screen. It is invoked in the Makefile after a TLS file has been created.

1. **Builds**

There are 2 different builds that encompass building the u-boot image with its IDS core and board-specific support objects as well as building standalone tests.

* 1. **U-boot Image Build**

The u-boot directory Makefile calls out 2 builds: one for the IDS Core objects and one for the TCM board-specific objects.

* + 1. **IDS Core**

The ../uboot/Makefile line “**LIBS += lib\_ids/libids.a**” builds the core IDS objects. It includes these source files in the following directories:

**../uboot/lib\_ids/**:

main.c, menu.c, menutab.c, cmd.c, builder.c, exec.c, fileio.c, flash.c, memtools.c, patgen.c, pgen.c, progress\_bar.c

**../uboot/lib\_ids/built-ins/:**

ids\_nand.c, ids\_tcm.c, ids\_ram.c

**../uboot/board/anacaz/ids/ram/:**

ram-addr-hi.c, ram-addr-lo.c, ram-alias.c, ram-refresh.c, ram-cache.c, ram-readback.c

**../uboot/board/anacaz/ids/lib/:**

reg-rw.c, reg-ro.c, reg-wo.c

* + 1. **TCM Board-specific**

The **../uboot/Makefile** line “**LIBS += board/$(BOARDDIR)/lib$(BOARD).a**” builds the TCM board-specific objects for IDS. The library name translates to “**libtcm.a**”. It includes these source files in the following directories:

**../uboot/board/anacaz/tcm/:**

tcm.c, nand.c, cmd\_gpio.c, cmd\_led.c, cmd\_fpga.c, cmd\_a6conf.c, cmd\_upgrade.c, cmd\_kupgrade.c, npm\_boot.c

**../uboot/board/anacaz/ids/lib:**

idshw\_anacaz.c

* 1. **Standalone Test Image Builds**

There is 1 directory for standalone TCM test builds.

* + 1. **TCM Standalone Tests**

All TCM board standalone test source files are contained in the following directory:

**../uboot/board/anacaz/ids/tcm/**

There are 18 source files which, when built by the Makefile, are used to create 431 IDS (individual test) files. Tests that use the same object are differentiated by the parameters specified in the Makefile using the “mktest” tool (see 6. for a description). The other tool used by Makefile is “mklist” which groups individual tests into test lists where specific hardware is targeted. Each test list can have a maximum of 15 tests. The Makefile creates 60 TLS files from the 431 IDS files. The 18 files are:

a6-arm-reset.c, a6-config.c, a6-dump.c, a6-install.c, a6-load.c, a6-memtest.c,

a6-poke.c, a6-reset.c, a6-runchk.c, a6-stress-ld.c, a6-verify.c, fpga-load.c,

hub-dump.c, hub-memtest.c, i2c.c, mux-load.c, mux-memtest.c, upma-config.c

Additionally, this Makefile also compiles, and uses in this build, the following source files from the **../uboot/board/anacaz/ids/lib/** directory:

ids\_jmpvec.c, reg-ro.c, reg-rw.c, reg-wo.c

Note that, while reg-ro and reg-wo are called out in the Makefile for building, they are never actually included in any test that the Makefile creates.

To build all TCM standalone tests, just do a “make” in this directory.

1. **Config Files**

The config files that are relevant for TCM uboot builds is described in the following section.

* 1. **../uboot/include/config.mk**

This config file contains the definitions for board-specific Makefile variables required for uboot builds: **ARCH**, **CPU**, **BOARD**, and **VENDOR**.

* 1. **Board specific config files**

More detailed build definitions for the TCM board are contained in a file in the configs directory at ../uboot/include/configs/: **anacaz\_tcm.h**.

These files provide definitions for things like system clock setup, system I/O configuration, GPIO configuration, etc.

1. **TCM Menus**

Menus are at the core of IDS. Everything is done through some type of menu. There are menus for TLS file lists, menus for IDS test lists and for individual tests. There is a TFTP download menu, an execution control menu and a master list menu from which downloaded tests are executed. Menus fall into 3 categories: core menus that are created at compile time; core menus that are dynamically built at runtime; and test menus that are also built at compile time. Note that menus that reflect standalone tests, i.e., tests built outside of uboot, are dynamically built at run time.

Important note: all menus that contain TLS files, other than the **Mode Menu**, allow a maximum of 15 entries. The header file definition for a menu explicitly provides for 16 entries, but, in the menus under discussion here, the first entry is always “**1) RUN**”. That leaves 15 locations available to hold either TLS entries or IDS (individual tests) entries.

Limiting the size of menus to 16 entries was done to limit the amount of target board memory that would be consumed by IDS. A possible modification to this rigid approach would be to define a size global in the board config file that, depending on the amount of memory available, could be increased to a larger value, probably in power of 2 increments.

* 1. **Core menus – Compile time**

These are core menus that are created at compile time. They are the means to controlling the IDS process flow.

* + 1. **Main Menu**

This menu is the entry point to IDS from uboot. At the uboot prompt, the user types the command “**ids**” followed by a return. This action brings up the top level menu in IDS. It’s both the access point from uboot to IDS and the final stage in the graceful return from IDS back to uboot. It presents 3 choices: **DATE** (set date), **TIME** (set time) and **IDS** which starts IDS menu processing. For now, **DATE**  and **TIME** don’t do anything useful.

* + 1. **Mode Menu**

This menu is the mechanism used to provide the user access to menus containing lists of TLS files that are functionally related, i.e., perform services or execute tests on related hardware. The individual TLS menu lists, test menus that are created at compile time, are described in section 8.3, below. The following entries in the Mode Menu that are not covered in section 8.3 are described here.

1. **AUTORUN**

This selection provides the means to allow a single menu selection to process a set of TLS lists through the test cycle. These lists are hard coded in the function “**regress\_run\_appl()**” in **menu.c**. While any number of pre-defined TLS lists can be included here, currently only 3 lists are defined: **file\_mgr\_a6ddr\_menu**, **file\_mgr\_fpga\_menu** and **file\_mgr\_chan\_menu**. Notice that these 3 lists are among those covered in section 8.3.

1. **DEBUG**

This level provides the means to hard code menus into the uboot build. Its primary purpose is to provide tests and tools for use when tftp download is not yet available.

**11) PROGRESS BAR**

The progress bar is not fully implemented at this time.

**12) RTC TEST**

This selection is not fully implemented at this time

**13) master-list MENU**

This menu is dynamically built during TLS file download. The menu serves as the top of the menu tree that is built and reflects all menu levels for the TLS files selected. The user can traverse the menu tree and execute all tests or some subset of the tests included.

Note that when traversing the menu tree in the forward direction to get to the download menu, this **master-list MENU** is not included in that forward path. However, after TLS files are downloaded, this menu is inserted into the return path flow to give users the ability to execute the tests without returning all the way to the **MODE MENU**.

* + 1. **Execution Menu**

This menu is accessed anytime the **RUN** selection is made at any level starting from the **master-list MENU**. At the conclusion of the test run, this menu is re-accessed. The selections in this menu are:

1. GO - execute the test or tests from this menu tree level
2. SHOWLIST - placeholder, not yet implemented
3. NAME - shows the name of the current list/test being executed
4. CYCLES - user can set number of test cycles; default=1 cycle
5. ELAPSED TIME - shows elapsed execution time
6. VIEW LOGS - lets user review the logs of the tests executed
7. CLEAR LOGS - clears the logs
8. STATISTICS - shows completion statistics for all tests in current list
   * 1. **RAM Carve Menu**

This menu shows how memory has been carved out for use by IDS. The regions selected are:

1. U\_BOOT = 0x00200000 - uboot vector table stored here
2. MENU POOL = 0x00201000 - runtime menus built here
3. LOG FILES = 0x00300000 - test log file storage
4. RAM = 0x00400000 - scratch memory region
5. TLS FILES = 0x10000000 - region where TLS files are downloaded
6. SYSTEM = 0x17000000 - uboot upper memory region
   * 1. **TFTP File Menu**

This menu is the mechanism by which tftp downloads are performed. The various selections represent environment variable information needed to successfully access the target tftp server and download the specified files. When the initial selection **GO** is selected, the appropriate uboot environment variables are updated from this menu which permits the download to be executes successfully. Prior to executing the download, the user can change any of the selection data. The selections are:

1. GO - starts the download process
2. METHOD - tftp
3. GATEWAY IP - local gateway address
4. NETMASK - appropriate net mask
5. HOSTNAME - server host name, not used, set to <NULL>
6. ROOT PATH - TLS file root path, not used, set to <NULL>
7. IP ADDR - ip address of target board
8. SERVER IP - ip address of tftp server
9. DNS IP - DNS ip address, not used, set to <NULL>
10. NIS DOMAIN - unknown, set to <NULL>
    * 1. **Xmodem Menu**

This menu is defined in the IDS core source code, but its functionality has not been implemented.

* 1. **Core menus –Run time**

These are core menus that are created at run time. They provide support for running tests and keeping logs of test activity.

* + 1. **Log Menu**

Each time the **RUN** selection is used to run all test lists, a single test list or a single test, a log file of the test activity is created. The log file is named for the menu from which the **RUN** command was selected. Additionally, to ensure differentiation between log file names, each log file has a ‘:’ and sequentially increasing number attached to it. For example, running the following menus results in the corresponding log file names:

MENU LOG FILE

master-list menu = 1) master-list:1

master-list menu = 2) master-lilst:2

a6-stress-1-3 menu = 3) a6-stress-1-3:3

This log menu can be viewed from the **EXECUTION MENU** (see section 8.1.3) by choosing the following selection:

1. **VIEW LOGS**

Once the log menu is displayed, the user can view the contents of any log by selecting the number preceding the log file name.

The log menu can be cleared from the **EXECUTION MENU** (see section 8.1.3) by choosing the following selection:

1. **CLEAR LOGS**
   * 1. **List Menu – Master List Menu**

This menu is dynamically built during the download process to represent the set of standalone tests that will be executed during test runs. It is described in some detail in other sections: **8.1.2 Mode Menu** and **9.2.2 Auto Menu Build**. It holds a maximum of 15 TLS file names with each TLS file representing up to 15 individual tests.

All execution of standalone tests starts at this menu. From here, the user can descend the menu tree to an individual TLS test list or down to an individual test where test parameters are displayed and can be modified. At every level, starting with this menu, the user can select **RUN** to perform all tests from that level down the remainder of the menu tree.

Currently, there are only 2 ways to clear this menu. First, during the download process, if this menu’s limit of 15 entries is exceeded, then the following message displays:

**Maximum number of test lists (15) already loaded!!!**

**Do you want to clear list? [y]**

When the user enters ‘y’ at the prompt, IDS clears the existing **List Menu** and populates it with the downloaded TLS file that triggered the message as well as all the remaining TLS files in the current download list.

Entering any other response to the message causes the download process to terminate leaving the **List Menu** intact as it currently exists with a full list of TLS files.

The second way to clear this menu is to back out of IDS all the way to the uboot prompt and then to re-enter IDS.

* + 1. **Debug List Menu**

While this menu is not created at compile time but at run time, it is populated by the IDS initialization process with pre-defined tests that are included in the uboot build. Currently, the menu is populated with a NAND test and 6 RAM tests:

1. NAND - list contains single NAND test

1) NAND TEST

1. RAM - list contains 6 RAM tests
2. RAM REFRESH LO
3. RAM INTEGRITY LO
4. RAM CACHE LO
5. RAM REFRESH HI
6. RAM INTEGRITY HI
7. RAM CACHE HI

Since this menu is entirely contained within the uboot build, it can also provide additional services. Service menu entries, such as memory peek and poke tools, could be added to this menu to provide board bringup capability prior to the availability of external access to standalone tests and services.

* 1. **Test menus – Compile time**

These menus appear in the **MODE MENU** (see section 8.1.2) and contain pre-defined TLS files that are functionally related, i.e., perform services or execute tests on related hardware. They are defined in the source file **menutab.c** and are part of the uboot build. They are a way to get around the lack of an extra menu layer that would represent a “list of lists”, i.e., multiple TLS lists contained within a single menu. A future upgrade to provide this extra menu layer support would allow this menu to be taken out of uboot and incorporate it into the standalone test build process.

Many of the individual IDS files that make up the TLS file entries don’t actually perform tests but provide a service such as setting TFTP environment variables and downloading FPGA images to local memory as well as downloading those images to the FPGAs. Additionally, there is A6 CODEC programming that is also downloaded to memory and then to the CODECs. It is important that the required FPGA images and CODEC programming are present in the server’s TFTP directory with file names consistent with those called out in the standalone test Makefile. Without them, downloads will fail.

* + 1. **File Manager Menu**

This menu is used to support 2 selections in the **MODE MENU**:

**4) INSTALLED TESTS**

**5) BRINGUP DVT**

It contains TLS files for tests that are used for regressions. Currently, there are 2 TLS files contained in this menu targeting specific hardware:

**ppc.tls**

**ram-lite.tls**

* + 1. **File Manager Config Menu**

This menu supports the following selection in the **MODE MENU**:

**3 ) CONFIG ONLY**

This menu contains TLS files that provide a service. They are designed to provide the mechanism by which the FPGA can be downloaded with appropriate firmware.

Currently, the TLS files pre-defined in this menu are not being built at any level, neither u-boot nor standalone. No Makefile has instructions to create them.

* + 1. **File Manager EDVT Menu**

This menu supports the following selection in the **MODE MENU**:

**6) ENVIRON DVT**

This menu contains TLS files used for EDVT. They include A6 Codec tests, MUX and HUB FPGA tests and memory tests:

**a6-stress-1.tls**

**a6-stress-2.tls**

**mux.tls**

**hub.tls**

**a6-runchk.tls**

**ram.tls**

* + 1. **File Manager a6addr Menu**

This menu supports the following selection in the **MODE MENU**:

**7) A6INTR TEST**

This menu contains TLS files that reset major board components and initialize them to operational states, test A6 CODEC registers, reset A6 CODECs and download A6 CODEC programs:

**tcm-reset.tls**

**a6-regs.tls**

**a600-intr.tls**

**a601-intr.tls**

**a602-intr.tls**

**a603-intr.tls**

**a604-intr.tls**

**a605-intr.tls**

**a606-intr.tls**

**a607-intr.tls**

**a608-intr.tls**

**a609-intr.tls**

**a610-intr.tls**

**a611-intr.tls**

* + 1. **File Manager FPGA Menu**

This menu supports the following selection in the **MODE MENU**:

**8) TCM START-UP**

This menu contains TLS files that test system memory access; test lbus registers that control lbus banks, upm, controller and flash; test RAM address and refresh tests; perform LED color tests; perform i2c initialization and target board reset; perform a variety of A6 CODEC tests and provide an A6 CODEC memory dump tool:

**ppc.tls**

**ram-lite.tls**

**i2c-init.tls**

**led-all.tls**

**tcm-reset.tls**

**fpga-init.tls**

**a600-tcmverify.tls**

**a601-tcmverify.tls**

**a602-tcmapp.tls**

**a604-tcmapp.tls**

**a606-tcmapp.tls**

**a608-tcmapp.tls**

**a610-tcmapp.tls**

**a6-dump.tls**

* + 1. **File Manager CHAN Menu**

This menu supports the following selection in the **MODE MENU**:

**9) TCM CHAN RESET**

This menu contains TLS files that perform channel reset sequences for all A6 CODECs including, per CODEC: turn off LED; hub stop, disable and enable; CODEC reset; download CODEC program; start hub; turn on LED; and perform a variety of RAM tests:

**ch00-reset.tls**

**ch01-reset.tls**

**ch02-reset.tls**

**ch03-reset.tls**

**ch04-reset.tls**

**ch05-reset.tls**

**ch06-reset.tls**

**ch07-reset.tls**

**ch08-reset.tls**

**ch09-reset.tls**

**ch10-reset.tls**

**ch11-reset.tls**

**ram.tls**

* + 1. **File Manager a6addr Debug Menu**

This menu supports the following selection in the **MODE MENU**:

1. **A6 DEBUG**

This menu contains TLS files that perform an i2c initialization; initialize the hub and mux FPGAs; configure A6 CODECs for 283 MHz; download A6 CODEC program and verify program was downloaded successfully:

**I2c-init.tls**

**fpga-init.tls**

**a6-config-283.tls**

**a6-tcmapp-rev2.tls**

**a6-tcmapp-rev2.tls**

**i2c-init.tls**

**a6-config-283.tls**

**a6-tcmapp-rev2.tls**

1. **Download Process**

The prelude to downloading TLS files begins with the Mode Menu (see section 8.1.2 above). It contains mostly lists of TLS list files plus the “**Debug**” and “**Autorun**” selections and the “**master-list Menu**”. This discussion focuses on the lists of TLS list files. Select any entry from 3 through 10. The selected entry’s list of TLS files is presented in a separate menu window. Note that the first entry is “**1) RUN**”. The remaining entries contain names of TLS files and their location in the TFTP directory on the server from which downloads are done. Selecting the “**1) RUN**” entry starts the download process by bringing up the TFTP File Menu.

* 1. **TFTP File Menu**

This menu (see section 8.1.5 above) contains all information needed to perform a TFTP download from the server including gateway ip, netmask, ip addr, server ip, etc. This information is extracted from the uboot environment variables at IDS initialization. This allows the user to view these variables and to change them, if necessary, prior to download.

Select the “**1) GO**” entry to begin the download process. IDS uses the tftp information in the menu to set up the corresponding uboot environment variables that support tftp transfers. IDS uses the uboot utility “**NetLoop**” to perform the download.

* 1. **Downloading TLS Files**

Each TLS file consists of a TLS header detailing information about all IDS files included in the TLS file followed by a sequence of IDS headers with parameters and associated executable images. The download process downloads each TLS file in turn as it appears in the selected menu list and stores it at a well known location in local storage. That well known location is contained in the “**RAM CARVE**” menu (see section 8.1.4 above).

* + 1. **TLS File Memory Allocation Storage**

Currently, downloaded TLS files are stored at local memory address 0x10000000. This can change as needed. The first TLS file in the file menu is loaded at this address. IDS maintains the load address of each TLS file and its size. The size value is used to calculate the load address for the next TLS file in the download list.

Note: as long as the number of TLS files downloaded to local storage does not exceed 15, additional TLS lists can be processed for download and for processing into the “**master-list menu**”.

For example, in the Mode Menu, where the download process begins, there are 2 entries that, together, contain 8 TLS files for download (well within the 15 entry limit). These entries are: “**5) BRINGUP** DVT” and “**6) ENVIRON** DVT”. Their respective TLS files could be downloaded to local memory and have their individual tests executed. There are a total of 8 TLS files with their associated tests for a total of 66 tests.

* + 1. **Auto Menu Build**

After a download completes, IDS automatically builds up an execution menu from the downloaded TLS files starting with the first one at address 0x10000000. The TLS header and all associated IDS headers are processed to build up the execution menu which is internally called “**list\_menu”** and which the user knows as the “**master-list MENU**” from the Mode Menu.

From our example in the previous section this would create a menu tree with menus for each TLS entry with its associated list of IDS test entries and finally each IDS test’s parameter list. The return path up the menu tree from the download menu gets an extra stop at the newly created “**master-list MENU**” which gives access to all menu levels: TLS list, IDS list and individual IDS test.

The first entry in each menu is “**1) RUN**” and can be selected to run all tests at that level and below. That would cause all 66 tests to be executed. Selecting a TLS entry would, instead, present a menu of all tests defined for that entry and all these tests, only, could be performed by selecting “**1) RUN**”. And, finally, selecting an individual test would bring up that test’s parameter menu which could be run as is or parameters could be modified before test execution.

1. **Test Execution Flow**

Every menu has as “1) RUN” as its first entry. Starting at the Master List Menu, selecting this first entry causes all tests specified at this level to be executed. At the Master List Menu, all TLS entries will be executed.

If, instead of selecting the **RUN** option, the user selects a TLS entry, that will bring up a list menu of all the tests contained in that TLS entry. Selecting the **RUN** entry at this level causes all the tests in this list, only, to execute.

Finally, if the user selects a single test entry, the resulting test menu will show all the parameters that serve as test arguments. Selecting the **RUN** entry will cause this test to execute.

To keep the discussion simple, the remaining subsections are described from the perspective of executing **RUN** from the Master List Menu. Note that, regardless of the level from which **RUN** is executed, the **Execution Menu** is the next stop in the test flow.

The **Execution Menu** provides execution control through various menu entries such as “**1) GO**” which actually sets the test flow in motion, “**4) CYCLES**” which sets the number of cycles the test run will execute, etc. The **GO** entry is initialized as type “ITEM\_APPL” and points to the function “**test\_go\_appl**” as the first function to execute. Selecting “**GO**” transfers control to this function.

Note that in most cases, while a test is executing, IDS samples the user input queue for a **CTRL + c** key combination. If detected, IDS stops test run and exits to menu processing.

* 1. **test\_go\_appl(menu\_t \*p)**

The argument passed in is a pointer to the menu at the level from which the RUN entry was selected. In this case, it’s the Master List Menu.

This function sets up the cycle count for the test run, creates a new log file in memory to capture this test run and calls the next function in the test execution flow, “**do\_test\_cycles\_appl**”.

Note: a dynamic log files menu is built and tracks test runs. Each run adds a log entry to the menu. This menu is accessed from the execution menu at the end of each test run by selecting “6) VIEW LOGS”. When selected, the menu shows all test runs of the master list with each run uniquely identified by a **“:<number>**” ending, i.e,. **master-list:1**, **master-list:2**, etc. Selecting a log entry causes the content of that log to be displayed at the serial console.

Upon return from the called function, the appropriate status message is queued up (fail text, abort text or pass text). The elapsed time is calculated and the date is fetched and stored in the Elapsed Time entry of the Execution Menu. The PASS/FAIL message is printed to the console, and, if a log file is open, the message is also printed to the log file. Finally the log file is closed.

* 1. **do\_test\_cycles\_appl(int cycles, int (\*test\_appl)(menu\_t \*p), menu\_t \*p)**

This function is the test cycle counter for the test run. It will repeat the tests either for as many cycles as have been selected at the start of this run or forever if the cycle count is set to 0. Besides calling the **do\_test\_go\_appl** function to execute the tests, this function also checks, at the conclusion of each cycle, for user input of the CTRL-C key combination which would abort the test run.

The arguments passed in are: the number of cycles to repeat all tests in this test run; a pointer to the next function to call, **do\_test\_go\_appl**; and a pointer to the Master List Menu.

* 1. **do\_test\_go\_appl(menu\_t \*p)**

This function will parse the menu passed to it to identify the next entry to process. The next entry will point to a TLS menu entry, so the function will recursively call itself passing a pointer to the TLS entry. When the recursively called function then processes the TLS menu entry, it will encounter one or more IDS menu entries where each entry represents a single test. Since this IDS entry is itself a list entry, this function will again recursively call itself passing a pointer to the IDS test entry. This brings processing to the lowest menu level, the individual test. Each entry in this menu represents an argument to the test. These arguments were built into the IDS entry by the “mktest” tool at compile time.

Initially, the argument passed to this function is a pointer to the Master List Menu. During subsequent recursive entries to this function, the arguments will point to a TLS menu or an IDS menu.

Once the function has reached down to the individual test level, it calls another function, **do\_test\_appl**, to actually process the specified test. It passes two arguments: a pointer to the parameter menu for the test and a pointer to a local variable through which the called function will return the “action” that should be taken at the conclusion of the test. The possible actions are: retry, next, skip, stop, abort, halt, spin, pause, ignore, back, wait, sleep or loop. See the section, below, for an explanation of each action.

A filtering of the action to take is done, and the appropriate condition is set to be performed. Finally, a check for CTRL-C is done to see if the user requested a test abort.

* 1. **do\_test\_appl(menu\_t \*p, char \*\*action)**

This function operates at the lowest level of the execution flow. It’s where the tests are actually executed.

The arguments passed in are: a pointer to the individual test menu which carries the test parameters and an indirect pointer to a resulting action that the calling function will use to decide how the test run should proceed.

The test function call arguments, **argc** and **argv**, are initialized from the test parameter list. Then the ELF executable for the test is booted from the image location in memory. The test executes and returns a value of either 0 (PASSED) or non-zero (FAILED). If the test failed, the **action** variable is set to point to the FAIL vector in the test header; if it passed, the PASS vector is placed in the **action** variable. Finally, the cycle count and date and time are output to the console and log file, if active. Note: test progress messages are output to the console and log file by the test itself.

* 1. **Pass/Fail Policy Actions**

At the conclusion of a test, there are a set of actions that can be taken depending on test result (**PASS** or **FAIL**) and pass/fail policy as set by the **mklist** utility during test build or as modified by the user at the last TLS menu which contains a list of tests and pass/fail policies for those tests. The default policy for **PASS** is **next** (execute the next test in the list) and for **FAIL** is **stop** (stop the test run and return to IDS menu processing). Note that for the **AUTORUN** selection in the **Mode Menu** that runs a pre-defined set of TLS files, the fail policy is to ignore the failure and continue with the next test in the list.

Note, in every test, regardless of the action specified, the user can input the **CTRL + c** key combination to cause IDS to end the test run and return to menu processing.

* + 1. **Edit Pass/Fail Policy**

When the user has traversed the menu tree to the last (lowest in the tree) TLS menu that shows a list of individual tests and their associated pass/fail policies, those policies can be modified by the user in this menu, only. Two CTRL + <key> combinations are defined for this purpose.

**CTRL + p**

This key combination directs IDS to allow changes to the **PASS** policies of any and all tests in the list. Entry of this key combination changes the “**ids>**” prompt at the bottom of the menu to “**pass>**”. In this new menu state, selecting any test entry causes IDS to present that entry’s **PASS** policy for editing. For example, if the following is an existing parameter in the menu:

**PASS FAIL CYCLES FAULTS**

1. **stress-setup-1.1 next stop 0 0**

then selecting this entry “**2**” displays the following:

**pass> 2 PASS = next**

with the cursor positioned under ‘**n**’ in **“next**”. The user then types the new policy followed by a <RETURN>. The newly entered policy replaces the original one in the menu and the “**pass>**” prompt is redisplayed to allow changes to other tests’ policies. Typing the **ESC** key or the ‘**u**’ key returns this menu’s state to normal with the “**ids>**” prompt displayed.

**CTRL + f**

This key combination directs IDS to allow changes to the **FAIL** policies of any and all tests in the list. Entry of this key combination changes the “**ids>**” prompt at the bottom of the menu to “**fail>**”. In this new menu state, selecting any test entry causes IDS to present that entry’s **FAIL** policy for editing. For example, if the following is an existing parameter in the menu:

**PASS FAIL CYCLES FAULTS**

1. **stress-setup-1.1 next stop 0 0**

then selecting this entry “**2**” displays the following:

**fail> 2 PASS = stop**

with the cursor positioned under ‘**s**’ in **“stop**”. The user then types the new policy followed by a <RETURN>. The newly entered policy replaces the original one in the menu and the “**fail>**” prompt is redisplayed to allow changes to other tests’ policies. Typing the **ESC** key or the ‘**u**’ key returns this menu’s state to normal with the “**ids>**” prompt displayed.

* + 1. **<selection:2-16>**

This action represents one of the possible 15 selections in this menu list. This action directs IDS to transfer test execution to the test at the selected list entry.

* + 1. **next**

This action directs IDS to execute the next test in the menu.

* + 1. **stop, abort, halt**

Currently, these actions direct IDS to stop the test run and return to menu processing.

* + 1. **ignore**

This action directs IDS to ignore the failure and continue with the next test in the menu.

* + 1. **skip**

This action directs IDS to skip the next test in the list and continue with the following test.

* + 1. **retry**

This action directs IDS to re-run the current test. The pass/fail status of the current test is not considered. The max number of retries is hardcoded to 3. When the retry count is exceeded, IDS stops the test run and returns to menu processing.

* + 1. **pause**

This action directs IDS to present the following message: **“Hit any key to continue…”**. IDS then waits for the user to enter any key and then continues testing with the next test in the menu.

* + 1. **back**

This action directs IDS to re-run the previous test in the menu. The pass/fail status of the current test is not considered. The end result is that testing will loop on the previous test and the current test

* + 1. **spin**

This action directs IDS to continuously run the current test.

* + 1. **wait**

This action directs IDS to delay processing for a specific time and then to continue with the next test in the menu. If, during the build process, the action entry for this test in the Makefile includes a number value, i.e., “**wait=12345**”, then the wait time is set to “**12345**” microseconds. If the wait entry in the Makefile has no associated number value, then the wait delay defaults to 1000 microseconds.

* + 1. **sleep**

This action directs IDS to delay processing for a specific time, in seconds, and then to continue with the next test in the menu. If, during the build process, the action entry for this test in the Makefile includes a number value, i.e., “**sleep=5**”, then the wait time is set to “**5**” seconds. If the wait entry in the Makefile has no associated number value, then the wait delay defaults to 1 second.

* + 1. **loop**

This action directs IDS to re-run the current test the specified number of times and then to stop testing and return to menu processing. This action is always in the **FAIL** track. If, during the build process, the action entry for this test in the Makefile includes a number value, i.e., “**loop=5**”, then the loop count is set to “**5**”. If the loop entry in the Makefile has no associated number value, then the loop count defaults to 1.

1. **U-boot Jump Vector Table**

In order to avoid having to develop a full library of utilities/drivers for standalone tests to use to access target board hardware and to provide software utilities, the mechanism of the u-boot jump vector table was created. It is designed to give standalone IDS tests access to u-boot utilities/drivers.

A well known location in local memory, which can be viewed in the “RAM CARVE” menu, was designated as the base address for an array of u-boot vectors which is populated by the IDS initialization code in IDS core with pointers to all relevant u-boot utilities and drivers.

* 1. **load\_u\_boot\_vectors()**

The IDS initialization function “**load\_u\_boot\_vectors()**” located in **main.c** in the **../uboot/lib\_ids/** directory is called by the “**ids\_start()**” function to populate the vector table. The following definition in **main.c** provides the access to the table:

**#define u\_boot\_vectors ((void (\*\*)(void))0x00200000) /\* ram carved address \*/**

An example of how the table is populated follows:

**u\_boot\_vectors[0] = (void (\*)(void))NetSendPacket;**

**u\_boot\_vectors[1] = (void (\*)(void))NetSetHandler;**

**.**

**.**

**u\_boot\_vectors[80] = (void (\*)(void))i2c\_init;**

**u\_boot\_vectors[81] = (void (\*)(void))i2c\_probe;**

**.**

**.**

**u\_boot\_vectors[87] = (void (\*)(void))i2c\_reg\_write;**

**.**

**.**

**etc.**

* 1. **ids\_jmpvec.h**

This header file is located at **../uboot/include/ids/**.

This file defines the macros that provide access to the utilities/drivers in the jump vector table. Each macro is named appropriately for the specific utility or driver with appropriate call parameters, and each macro is defined at an offset into the jump vector table that corresponds to the location in the table at which that specific utility/driver was pre-loaded by the IDS initialization code. This header file contains the same definition for the jump vector table for use in standalone test builds:

**#define u\_boot\_vectors ((void (\*\*)(void))0x00200000) /\* ram carved address \*/**

An example of how the table is defined in the header file is:

**#define NETSENDPACKET(pkt,len) (\*((void (\*\*)(volatile uchar \*,int))u\_boot\_vectors)[0])(pkt,len);**

**#define NETSETHANDLER(f) (\*((void (\*\*)(rxhand\_f \*))u\_boot\_vectors)[1])(f)**

**.**

**.**

**#define I2C\_INIT(speed,slaveadd) (\*((void (\*\*)(int,int))u\_boot\_vectors)[80])(speed,slaveadd)**

**#define I2C\_PROBE(chip) (\*((void (\*\*)(unsigned char))u\_boot\_vectors)[81])(chip)**

**.**

**.**

**#define I2C\_REG\_WRITE(a,r,v) \**

**(\*((void (\*\*)(unsigned char, unsigned char, unsigned char))u\_boot\_vectors)[87])(a,r,v)**

**.**

**.**

**etc.**

* 1. **ids\_jmpvec.c**

This file is located at **../uboot/board/Anacaz/ids/lib/**.

It contains all the translations for jump vector table accesses. It translates the normal function call sequence into a vector table access. For example, the driver call to perform an i2c register write:

**i2c\_reg\_write(dev, reg, val);**

is mapped into this function in this ids\_jmpvec file:

**void i2c\_reg\_write(unsigned char addr, unsigned char reg, unsigned char val)**

**{**

**I2C\_REG\_WRITE(addr, reg, val);**

**}**

The mapping becomes clear when you look at the example in the previous section, 11.2, for **I2C\_REG\_WRITE** which is defined as an indexed access to the uboot i2c driver code through the pointer at jump vector table offset **87**.