**Anacaz**

**Integrated Diagnostics System (IDS) Design 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**](#_Toc279501114)

[**1.1 Document Purpose 3**](#_Toc279501115)

[**1.2 What is Covered in this Document 3**](#_Toc279501116)

[**1.3 Hierarchical Description of IDS Work Flow 3**](#_Toc279501117)

[**1.3.1 Standalone Directory and Contents 6**](#_Toc279501118)

[**1.3.2 Build TLS Files 6**](#_Toc279501119)

[**1.3.3 Load TLS Files 7**](#_Toc279501120)

[**1.3.4 Run IDS Tests 9**](#_Toc279501121)

[**2. Build Tools 11**](#_Toc279501122)

[**2.1 mktest 11**](#_Toc279501123)

[**2.1.1 Structures 13**](#_Toc279501124)

[**2.1.2 Inputs 14**](#_Toc279501125)

[**2.1.3 Outputs 15**](#_Toc279501126)

[**2.2 mklist 16**](#_Toc279501127)

[**2.2.1 Structures 17**](#_Toc279501128)

[**2.2.2 Inputs 18**](#_Toc279501129)

[**2.2.3 Outputs 19**](#_Toc279501130)

[**3. Download Process 19**](#_Toc279501131)

[**3.1 Structures 19**](#_Toc279501132)

[**3.1.1 ids\_dir\_t 20**](#_Toc279501133)

[**3.1.2 Downloaded Files Layout in Local Memory 21**](#_Toc279501134)

[**3.2 Downloading TLS Files 22**](#_Toc279501135)

[**4. Menu Build Process 24**](#_Toc279501136)

[**4.1 Structures 24**](#_Toc279501137)

[**4.2 menu\_t 24**](#_Toc279501138)

[**4.3 item\_t 25**](#_Toc279501139)

[**4.4 Creating and Modifying Test Parameters 28**](#_Toc279501140)

[**4.5 Menutab.c 29**](#_Toc279501141)

[**4.6 Core IDS Source Files 29**](#_Toc279501142)

[**4.7 Walking the Menu Tree 29**](#_Toc279501143)

[**4.8 Building Menus from TLS Files 30**](#_Toc279501144)

[**4.9 Menu Layout in Local Memory 34**](#_Toc279501145)

[**7. How Test Execution uses Menus 37**](#_Toc279501146)

[**7.1 test\_go\_appl(menu\_t \*p) 37**](#_Toc279501147)

[**7.2 do\_test\_cycles\_appl(int cycles, int (\*test\_appl)(menu\_t \*p), menu\_t \*p) 38**](#_Toc279501148)

[**7.3 do\_test\_go\_appl(menu\_t \*p) 38**](#_Toc279501149)

[**7.4 do\_test\_appl(menu\_t \*p, char \*\*action) 39**](#_Toc279501150)

[**7.5 Pass/Fail Policy Actions 39**](#_Toc279501151)

[**7.5.1 Edit Pass/Fail Policy 40**](#_Toc279501152)

[**7.5.2 <selection:2-16> 41**](#_Toc279501153)

[**7.5.3 next 41**](#_Toc279501154)

[**7.5.4 stop, abort, halt 41**](#_Toc279501155)

[**7.5.5 ignore 41**](#_Toc279501156)

[**7.5.6 skip 41**](#_Toc279501157)

[**7.5.7 retry 41**](#_Toc279501158)

[**7.5.8 pause 41**](#_Toc279501159)

[**7.5.9 back 42**](#_Toc279501160)

[**7.5.10 spin 42**](#_Toc279501161)

[**7.5.11 wait 42**](#_Toc279501162)

[**7.5.12 sleep 42**](#_Toc279501163)

[**7.5.13 loop 42**](#_Toc279501164)

[**7.6 Core IDS Board specific issues 42**](#_Toc279501165)

[**8. Hot Key Processing 43**](#_Toc279501166)

[**8.1 Show “RAM CARVE” menu - ^T 43**](#_Toc279501167)

[**8.2 Modify “PASS” policy - ^P 44**](#_Toc279501168)

[**8.3 Modify “FAIL” policy - ^F 44**](#_Toc279501169)

[**8.4 Menu HELP - ^N 45**](#_Toc279501170)

[**8.5 TLS Download Control - ^L 45**](#_Toc279501171)

[**9. IDS Commands Processing 46**](#_Toc279501172)

# Overview

## Document Purpose

The purpose of this document is to provide an in-depth description of the basic processes, structures and tools IDS requires to perform its mission as a framework for executing diagnostic tests in a controlled environment. This document will provide the diagnostic software engineer the information required to understand the structure and function of IDS such that the engineer can make future changes/additions to the IDS software as needed.

## What is Covered in this Document

The components of IDS covered in this document start with the tools used in the board specific Makefile which builds standalone tests. The tools are “mktest” and “mklist” which, together, build the IDS and TLS files that are downloaded to the target board. Next, the document details the download process, the automatic menu build process and, finally, how IDS uses the menus during test execution.

## Hierarchical Description of IDS Work Flow

The following sub-sections provide an hierarchical depiction of the flow from source creation to test execution.

The overall hierarchical depiction is:

Build TLS Files

Load TLS Files

Run IDS Tests

The first block encompasses creating standalone test source files, converting them to IDS and TLS files and copying TLS files to the tftp server. The second block describes the process, via IDS menus, of downloading TLS files from the tftp server to the target board. And, the third block details how to use IDS menus to run standalone IDS tests.

## Standalone Directory and Contents

All IDS standalone test development is done outside of the IDS core directories and code. A single standalone directory contains the files necessary for building the standalone diagnostic tests. The TCM standalone build directory is at:

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

For the TCM board, when the branch is first pulled from SVN, the standalone directory has one Makefile, 19 source (.c) files and a zip file that contains the 4 binaries that various tests download to the TCM board during testing.

The Makefile uses these 19 source files to build 443 individual IDS test files (.ids). Out of these IDS files, 61 TLS files, containing between 1 and 15 IDS test files, are built. It is these TLS files that are copied to the C:/tftp/tcm directory from where they are downloaded to the TCM board under control of the IDS core image running on the TCM board.

The individual source (.c) files are built into object (.o) files. It’s these .o files that the Makefile builds multiple tests with unique names. A single .o file can be the source for multiple .ids test files. Differentiation between IDS files is effected by the parameters the Makefile specifies for each unique instantiation of the .o object file into IDS files.

See section **2.0** and its subsections for a full description of this process.

## Build TLS Files

This diagram describes the flow from creating standalone tests through compilation, IDS file creation from which TLS files are created and copying resulting TLS files to the tftp server.

Create IDS .c standalone Tests

**i2c.c**

Makefile:

Compile tests into ELF loadable.o files

**eg: ../uboot/board/Anacaz/ids/tcm/**

**i2c.c i2c.o**

Makefile:

Run mktest, turn .o files into .ids files

**mktest: i2c.o i2c-init-0.ids**

**mktest: i2c.o i2c-probe.ids**

Makefile:

Run mklist, turn .ids files into .tls files

**mklist: i2c-init-0.ids, i2c-probe.ids**

**i2c-init.tls**

Copy .tls files to tftp directory of tftp server

**copy/paste from build dir c:/tftp**

## Load TLS Files

This diagram describes the flow from bring up of IDS on the target board through download of the TLS files and the autobuild of menus using the downloaded TLS files.

Bring up IDS on target board via serial window

**At u-boot prompt: => ids <return>**

Select TLS list from Mode Menu & walk menu tree to TFTP download menu

**Mode menu item: 4) INSTALLED TESTS**

**User selects item: Ids> 4 <return>**

**Installed Tests Item: 1) RUN**

**User selects item: ids> 1 <return>**

Download TLS files to dedicated local TLS file storage

**TFTP File menu item: 1) GO**

**User selects item: ids> 1 <return>**

IDS autobuilds menus from TLS files to dedicated local menu storage

**Menus built after TLS file download complete**

**When done, return to TFTP File menu**

Master-list Menu filled out with executable TLS lists

**Master List Items: 1) RUN**

**2) ppc-1 *TLS List***

**3) ram-lite-2 *TLS List***

## 

## Run IDS Tests

This diagram describes the flow from selecting the TLS list through executing one or more tests

.

Walk menu tree to Master-list Menu

**Master List Items: 1) RUN**

**2) ppc-1 *TLS List***

**3) ram-lite-2 *TLS List***

Select action:

1)All TLS test lists **Select “1) RUN” to execute all TLS lists**

2)Single TLS test list **Select “2) ppc-1” to access single TLS list**

**Then select “1) RUN” in that list to execute all list tests**

3) Single IDS test **Select “6) lbus-upm-32-1.5” to access single IDS test menu**

**Then select “1) RUN” in IDS test menu to execute test**

Execution CTRL Menu:

- alter test cycle count **Select “4) CYCLES” to modify test cycle count**

- GO starts test execution **Select “1) GO” to execute the test**

IDS ELF-loa6ds test binary from TLS Storage & runs test

Test executes; test progress output to serial window & log file in local memory

**[ TEST lbus-upm-32-1.5 : E0005068:E00050A0 32-BIT ACCESS ]**

**[ lbus-upm-32-1.5 REGISTER E0005068:=00000000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005070:=08040000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005074:=00000000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005078:=00000000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005084:=00000000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005088:=00000000 ]**

**[ lbus-upm-32-1.5 REGISTER E0005090:=00000001 ]**

**[ lbus-upm-32-1.5 REGISTER E00050A0:=00000000 ]**

**[ CYCLE 1:1 Thu Jan 01 01:33:32 1970 \*\*\*\* PASSED \*\*\*\* ]**

Test ends; control returns to Execution CTRL Menu:

- view log files

**Execution Menu Item: 6) VIEW LOGS**

**User Selects Item: ids> 6 <return>**

- view statistics

**Execution Menu Item: 8) STATISTICS**

**User Selects Item: ids> 8 <return>**

- clear logs

**Execution Menu Item: 7) CLEAR LOGS**

**User Selects Item: ids> 7 <return>**

Walk menu tree back to last list level (TLS or IDS); Select next test(s) to run

**User uses ‘u’ or ‘U’ or ESC key to move back up through the menus**

**Return to previous IDS test list to make next test selection:**

**1) RUN PASS FAIL CYCLES FAULTS**

**2) sys-ro-config-32-1.1 next stop 0 0**

**3) sys-rw-config-32-1.2 next stop 0 0**

**4) lbus-banks-lo-32-1.3 next stop 0 0**

**5) lbus-banks-hi-32-1.4 next stop 0 0**

**6) lbus-upm-32-1.5 next stop 3 0**

**7) lbus-ctrlr-32-1.6 next stop 0 0**

**8) lbus-flash-32-1.7 next stop 0 0**

**ids>**

# Build Tools

This section describes the 2 primary tools used by the Makefile in the standalone tests directory to produce the individual test IDS files and associated header (.h) files and to use the IDS and header files to produce the TLS files that are downloaded to the target board.

The Makefile in the target board build directory for standalone tests sets up 2 variables that reference the individual tool executable for use in creating .ids and .tls files:

**$(MKTEST)** translates to: **../uboot/tools/ids/mktest**

**$(MKLIST)** translates to: **../uboot/tools/ids/mklist**

## mktest

This tool’s function is to produce 2 output files: an .ids file with test header and elf loadable test binary and anuuu .h header file containing the arguments called out by the Makefile. The **mktest** tool opens 3 files: the object (.o) test file produced by the build process for read only and 2 read/write files: an IDS file with .ids extension and a header file with a .h extension. The respective handles are **orgfd** for the object file, **idsfd** for the IDS file and **hdrfd** for the header file.

Note that there is only one ‘**-**‘ delimited argument option for **mktest** in the Makefile. That option, “**-o**”, identifies the next parameter as the name for the output .ids file. In the Makefile example, below, “**-o mux-rw-16**”, results in a file named “**mux-rw-16.ids**”. The output file name defined by the “**-o**” option is also used to create the filename for the header file that is created, i.e., “**mux-rw-16.h**”.

After opening all files, **mktest** first writes the content for the header file. It outputs the standard header file preamble to assure that this header file is only included once in a compile. It then adds a warning not to edit this file. Here is an example using the **reg-rw.o** binary file as a starting point which is renamed for output into **mux-rw-16.ids** and **mux\_rw\_16.h** by **mktest**. After the first steps described here, the header file contains:

#ifndef \_mux\_rw\_16\_h

#define \_mux\_rw\_16\_h

/\*

\* WARNING! DO NOT EDIT -- THIS IS AN AUTO GENERATED FILE

\*/

Then **mktest** parses through the arguments passed in by the Makefile adding the following to the header file (again, using **mux\_rw\_16.h** as an example). The Makefile looks like this:

$(MKTEST) reg-rw -o mux-rw-16 access=short \

vers-0={0x80000000:0x0000} \

ctrl-2={0x80000002:0x0000} \

.

.

.

addr-A={0x8000000A:0xFFFF} \

data-C={0x8000000C:0xFFFF}

Each complete argument is space delimited. These arguments, starting with “access=short”, are written to **mux\_rw\_16.h** as:

#define ACCESS short

#define VERS\_0\_REG 0x80000000

#define VERS\_0\_MSK 0x0000

#define CTRL\_2\_REG 0x80000002

#define CTRL\_2\_MSK 0x0000

.

.

.

#define ADDR\_A\_REG 0x8000000A

#define ADDR\_A\_MSK 0xFFFF

#define DATA\_C\_REG 0x8000000C

#define DATA\_C\_MSK 0xFFFF

Note that the register definitions, which, in the Makefile appear together separated by a ‘:’, in the header file, are broken up into 2 contiguous lines: register address on the first line and register read/write mask on the second. Note also that the first argument in the Makefile, “access=short”, identifies this argument list as a set of register writes and reads. Specifically, “access” in the first argument position is the trigger.

The IDS test header contains an argument  **argc** count field and a 2-dimensional (15 x 32) **argv** vector array which permits up to 15 arguments per test and with a limit of 31 characters per argument (the 32nd position contains a string ending null character).

For each argument in the Makefile argument set, the complete argument, as defined in the Makefile, is copied to the next available array location and the **argc** count field is incremented. When the argument parsing is completed, the **argc** field will have the exact count of parameters and the **argv** vector array will contain all arguments in Makefile definition order. This is the order in which the arguments appear in the menus.

After **mktest** finishes processing all Makefile arguments into the IDS test header structure, that structure is output to the IDS file handle **idsfd.**

Next, **mktest** reads the binary file in 64k chunks and writes the binary to the IDS file immediately following the test header. In the Makefile example, above, the **reg-rw** element following **$(MKTEST)** refers to the file **reg-rw.o** which was built as an ELF executable binary file.

Finally, the ending string **“#endif /\* !\_mux\_rw\_16\_h \*/** “ is written to the header file and all 3 files, binary, header and IDS, are closed.

Note: the .h header file, while produced by **mktest**, is not used at this time by IDS but is intended for future development.

## Structures

IDS provides the following structure definition for the header in an IDS file:

**typedef struct test\_hdr**

**{**

**char name[32]; /\* Test name \*/**

**char pass[32]; /\* Pass vector \*/**

**char fail[32]; /\* Fail vector \*/**

**char size[8]; /\* Size in bytes including payload \*/**

**char cntr1[8]; /\* Changed from appl[16] 06-01-2009 \*/**

**char cntr2[8]; /\* Changed from appl[16] 06-01-2009 \*/**

**char argc[4]; /\* Argument count \*/**

**char argv[15][32]; /\* Argument vector \*/**

**#ifdef COMMENT**

**char help[256]; /\* Help Message (256 bytes) \*/**

**char \_[56]; /\* PAD to 768 bytes \*/**

**#endif /\* COMMENT \*/**

**} test\_hdr\_t;**

This structure is located in the file: **../uboot/include/ids/idsfs.h**

This structure is the one discussed in the previous sub-section as the test header. The full header, not including the definitions bracketed by **“#ifdef COMMENT**” and **“#endif** /\* **COMMENT** \*/”, is 604 bytes in size.

**mktest** fills out the following header elements based on the information provided by the Makefile call to this tool: **name**, **pass**, **fail**, **cntr1**, **cntr2**, and for **argc** and **argv,** as many arguments as Makefile specifies.

The layout of an IDS file created by **mktest** is:

|  |  |  |
| --- | --- | --- |
| **Start Offset** | **Size(Bytes)** | **Name** |
| 0x0 | 32 | Test Name |
| 0x20 | 32 | Pass Vector |
| 0x40 | 32 | Fail Vector |
| 0x60 | 8 | Size(bytes) including payload |
| 0x68 | 8 | Counter 1 (num of tests that PASSed per test run) |
| 0x70 | 8 | Counter 2 (number tests that FAILed per test run) |
| 0x78 | 4 | argc – argument count |
| 0x7C | 480 | 2 dimensional argument vector array: 15 x 32 |
| 0x25C | variable | ELF-executable Binary |

The IDS header is a fixed size of 604 bytes. The ELF-executable binary is variable. The Pass and Fail Vectors are described in detail later in this document in section **7.5.**

## Inputs

The inputs to **mktest** are provided by the **$(MKTEST)** Makefile instruction. These are:

< object file name minus the .o extension >

< -o > identifies next argument as a renaming of the executable for output

< renamed output filename > base for .ids and .h output filenames

< test parameter argument 1 >

< test parameter argument 2 >

.

.

< test parameter argument n >

The maximum number of test parameter arguments allowed is 15. Each test parameter argument cannot exceed 31 characters. The reason for this is that the IDS test header argument vector array allows 32 characters per argument, but it reserves the last character for the string ending null.

Here are 2 examples of Makefile invocations of **$(MKTEST)**; the first is a typical test; the second is a register-based test:

**$(MKTEST) i2c -o i2c-init-0 operation=init \**

**speed=400000 bus=0 clkdiv=0x2B**

**$(MKTEST) reg-rw -o mux-rw-16 access=short \**

**vers-0={0x80000000:0x0000} \**

**ctrl-2={0x80000002:0x0000} \**

**door-4={0x80000004:0x0000} \**

**door-6={0x80000006:0x0000} \**

**addr-8={0x80000008:0xFFFF} \**

**addr-A={0x8000000A:0xFFFF} \**

**data-C={0x8000000C:0xFFFF}**

It’s important to note that the order in which the parameters are defined in the Makefile are exactly the order in which they will appear in the IDS menu. The label strings to the left of the ‘**=**’ sign represent the parameter name, in this case a register, that appears on the left side of an IDS menu. The strings to the right of the ‘**=**’ appear on the right side of an IDS menu and are the initial test values for the test.

The “**access**” parameter is strictly used to describe register parameters. Specifically, the string to the right of the ‘**=** represents the size of the register reads and writes, namely:

char = 8 bits

short = 16 bits

long = 32 bits

longlong = 64 bits

The **mask** portion of the register parameter, that is, the string to the right of the ‘**:**’, must represent a value consistent with the **access** type.

## Outputs

**mktest** produces 2 output files: <output filename>.ids and <output filename>.h.

The header file, <output filename>.h, contains a list of all test parameter arguments in an individual test (see the 2.1 for a description). This file is not currently used by IDS but is, instead, intended for future IDS development.

The IDS file, <output filename>.ids, starts with an IDS header as defined in the “**Structures**” section previously for this tool. Following the header is the full ELF- executable binary from the object file.

## mklist

This tool’s function is to produce 1 output file: a .tls file containing a TLS header followed by as many IDS files as are called out by the Makefile. The **mklist** tool opens 2 files at any one time: the output .tls file which is open throughout the process and individual .ids files which are opened in the order in which they are called out by the **$(MKLIST)** command in the Makefile. The IDS files are opened for read only, processed and then closed so that no more than 1 IDS file is open at a time. The respective handles are **idsfd** for any input IDS file and **tlsfd** for the output TLS file.

Note that there is only one ‘**-**‘ delimited argument option for **mklist** in the Makefile. That option, “**-o**”, identifies the next parameter as the name for the output .tls file. In the Makefile example, below, **-o a600-tcmverify.tls**”, results in an output file of that name.

Prior to opening the files, **mklist** parses the argument list to ensure the minimum set of arguments are provided. Next, the TLS file is opened for Read/Write. The TLS filename is written to the TLS header, and the TLS header is written to the file.

Then **mklist** parses through the set of IDS file arguments from the Makefile, one IDS file at a time. If **pass** or **fail** arguments are present, they are saved for later updates to the specific IDS test header. **mklist** then opens the specified IDS file for read only. It performs an “fstat” function on the file to get the IDS file size for use in processing the IDS file into the TLS file. The IDS test header is read in from the IDS file. The filename, file size, pass and fail elements of the IDS test header are updated, and the header is written to the TLS file. Next, the ELF-loadable binary is copied from the IDS file to the TLS file. Finally, the IDS file is closed. **mklist** then continues processing with the next IDS file arguments in the Makefile argument list.

Throughout processing of IDS files, **mklist** keeps track of the count of IDS files and the total size of IDS files copied in to the TLS file. The final step in **mklist** takes is to update the TLS file with the IDS file count and the total file accumulated IDS tests file size. Then the TLS file is closed.

## Structures

IDS provides the following structure definition for the header in a TLS file:

**typedef struct list\_hdr**

**{**

**char name[32]; /\* Test list name \*/**

/\*

\* "IDS\0", -- currently this but maybe ...

\* "\001\002\003\004" -- something like this

\*/

**char magic[4]; /\* Test list identifier \*/**

/\*

\* CAUTION!!! Because of the limitation of the menu

\* structure in the header file src/menu.h for only

\* 16 items and one of the items has to be the "RUN"

\* item; there can only be 15 tests max!!!

\*/

**char tests[4]; /\* Number of tests in test list \*/**

**char size[16]; /\* Size in bytes including payload \*/**

/\*

\* Change to counter 06-01-2009

\*/

**char cntr[8]; /\* PAD to 64 bytes \*/**

**} list\_hdr\_t;**

The structure is located in the file: **../uboot/include/ids/idsfs.h**

**mklist** fills in the  **name** field from the output filename argument provided by the **$(MKLIST)** Makefile command. It places the string “IDS” into the **magic** field. It also updates the **tests** field from the cumulative IDS file count maintained throughout processing and the **size** field from the cumulative IDS file sizes maintained throughout processing.

The layout of a TLS file created by **mklist** is:

|  |  |  |
| --- | --- | --- |
| **Start Offset** | **Size(Bytes)** | **Name** |
| 0x0 | 32 | Test List Name |
| 0x20 | 4 | Test List Identifier |
| 0x24 | 4 | Number of tests in list |
| 0x28 | 16 | Size(bytes) including payload |
| 0x38 | 8 | Counter (pads structure to 64 bytes) |
| 0x40 | <size 1> | IDS Test File 1:  IDS Test Hdr + ELF Bin |
| 0x40 + <size 1> | <size 2> | IDS Test File 1:  IDS Test Hdr + ELF Bin |
| 0x40 + <size 1> +<size 2> | <size 3> | IDS Test File 3:  IDS Test Hdr + ELF Bin |
| . | . | . |
| . | . | . |
| 0x40 +  <prev sizes> | <size n-1> | IDS Test File <n – 1>:  IDS Test Hdr + ELF Bin |
| 0x40 +  <prev sizes> +  <size n-1> | <size n> | IDS Test File <n>:  IDS Test Hdr + ELF Bin |

The format of the IDS header is found in the **Structures** sub-section described previously. The TLS header portion is a fixed size of 64 bytes. The IDS test file sizes are variable.

## Inputs

The inputs to **mklist** are provided by the **$(MKLST)** Makefile command. These are:

< -o > identifies next argument as the output file name

< output TLS filename >

[< pass policy argument >]

[ < fail policy argument >]

< IDS filename 1 >

[< pass policy argument >]

[ < fail policy argument >]

< IDS filename 2 >

.

.

< IDS filename n >

The maximum of IDS files allowed as arguments is 15. Each test parameter argument cannot exceed 31 characters. The arguments can occur in the same file line but must be space delimited. Here is an example of a **$(MKLST)** Makefile invocation:

**$(MKLIST) -o a600-tcmverify.tls \**

**a600v-reset.ids \**

**-fail=stop a600v-ddr-337.ids \**  - note: 2 arguments in this line

**a600v-setup.ids \**

**a600v-copy.ids \**

**-fail=wait a600v-verify.ids \**

**-pass=sleep=3 a600v-start.ids \**

**-pass=skip -fail=loop=4 a600v-runchk.ids \** - 3 arguments here

**-pass=2 a600v-led-red.ids \**

**a600v-led-grn.ids**

The pass and fail vectors are described in detail later in this document in section **7.5.**

< What do the arguments pass and fail mean? >

Describe in which menu pass/fail appear and what they mean.

Fully described in Functional Spec., insert reference to appropriate section in that document.

## Outputs

**mklist** produces 1 output file: a TLS file (extension is .ids).

It starts with a TLS header as defined in the “**Structures**” section, described previously, for this tool. Following the header is the set of IDS files called out by the Makefile during build.

# Download Process

Currently, the download process starts at the **Mode Menu** which contains the set of 8 TLS lists as entries. These TLS lists are predefined and created as part of the core IDS build. The current set of downloadable TLS entries runs from menu entry 3 through entry 10. There is a detailed description of all the TLS lists in the **Integrated Diagnostic System Functional Specification**. In this document we will concentrate on the **TFTP File Menu** via which downloads are initiated and what happens when download is initiated.

## Structures

There are several structures that are common across menus and provide the support needed to execute the download process. The structures are **menu\_t**, **item\_t**, **idstime\_t, hwreg\_t** and **idsfs\_t**, and they are located in:

**../uboot/include/ids/menu.h**

**menu\_t** is the primary menu structure, and the others are in a supporting role, i.e., these subordinate structures are used in the **menu\_t** definition. The main structures, **menu\_t** and **item\_t** are described in the **MENU BUILD PROCESS** section, below. The others are simple enough to understand and can be found in the header file noted above.

There is one other very important structure that is defined in **../uboot/ids\_lib/fileio.c** and is described in the next sub-section.

## ids\_dir\_t

The **dir\_tbl[]** is an array of structures of type **ids\_dir\_t**. Its purpose is to maintain global information on the 3 types of memory allocations that IDS performs: TLS download files, menus and log files.

The structure definition is located in **../uboot/include/ids/idsfs.h**:

**typedef struct ids\_dir**

**{**

**void \*linkp; /\* Linked reference pointer \*/**

**unsigned long base\_addr; /\* From the GET\_RAM() macro \*/**

**unsigned long load\_addr; /\* Current available location \*/**

**unsigned int entries; /\* Number of files (itemcount) \*/**

**unsigned long** limit\_addr**; /\* Last address + 1 \*/**

**} ids\_dir\_t;**

The **dir\_tbl[]** array definition is in **../uboot/ids\_lib/fileio.c**:

**ids\_dir\_t dir\_tbl[] =**

**{**

**{ 0 }, /\* UBOOT \*/** *0x00200000*

**{ &list\_menu, 0, 0, 0, 0 }, /\* MENUPOOL \*/** *0x00201000*

**{ &log\_menu, 0, 0, 0, 0 }, /\* LOGFILES \*/** *0x00300000*

**{ 0 },** /\* RAM \*/*0x00400000*

**{ &file\_mgr\_menu, 0, 0, 0, 0 }, /\* TLSFILES \*/** *0x10000000*

**{ 0 }** /\* SYSTEM \*/*0x17000000*

**};**

The 3 relevant entries shown above are set to initial values by the function **fileio\_init()** which is called during IDS initialization. It’s located in **../uboot/lib\_ids/fileio.c**.

Initialization for the 3 relevant array entries sets the **base\_addr** and **load\_addr** fields to the start of the respective memory and the **entries** field to 0. During download (TLSFILES) and menu build (MENUPOOL), these 3 fields provide critical information for the respective processes. For example, during download, **base\_addr** always points to the start of download memory. **load\_addr** is used to locate the next available memory location during download. And **entries** keeps track of the total number of TLS files downloaded.

The **limit\_addr** address is only used for log file processing. The **linkp** pointer is set in the array definition but never used.

* + 1. **Downloaded Files Layout in Local Memory**

Following is the physical layout of downloaded TLS files in designated local memory. The starting address in local memory of downloaded TLS files is 0x1000000. The layout doesn’t include addresses since the size of the ELF Binaries is unknown. As an example, it incorporates 2 TLS files into the layout.

|  |  |  |
| --- | --- | --- |
|  | **Size(Bytes)** | **Name** |
| TLS File 1 | 64 | TLS Header 1 |
| IDS File 1 | 604 | IDS Header 1 |
|  | <ELF-size 1> | ELF Binary 1 |
| IDS File 2 | 604 | IDS Header 2 |
|  | <ELF-size 2> | ELF Binary 2 |
| . | . | . |
| . | . | . |
| IDS File 8 | 604 | IDS Header 8 |
|  | <ELF-size 8> | ELF Binary 8 |
| TLS File 2 | 64 | TLS Header 2 |
| IDS File 1 | 604 | IDS Header 1 |
|  | <ELF-size 1> | ELF Binary 1 |
| IDS File 2 | 604 | IDS Header 2 |
|  | <ELF-size 2> | ELF Binary 2 |
| . | . | . |
| . | . | . |
| IDS File 5 | 604 | IDS Header 5 |
|  | <ELF-size 5> | ELF Binary 5 |

## Downloading TLS Files

While getting to the point where download can be initiated requires traversing menus starting at the **Mode Menu**, the final menu in the tree which controls downloads is the **TFTP Menu**:

**1) GO**

**2) METHOD = tftp**

**3) GATEWAY IP = 10.0.3.1**

**4) NETMASK = 255.255.252.0**

**5) HOSTNAME = <NULL>**

**6) ROOT PATH = <NULL>**

**7) IP ADDR = 10.0.3.11**

**8) SERVER IP = 10.32.12.35**

**9) DNS IP = <NULL>**

**10) NIS DOMAIN = <NULL>**

**ids>**

This menu lets the user view, and change if needed, the download settings. Selecting the “**1) GO**” selection initiates the download process.

Menu display and user input (**ui()**) is processed via the **menu\_input()** function. With “**GO**” selected by the user, processing is picked up by **item\_input()**.

**item\_input()** takes 2 arguments: a pointer to the **tftp\_menu** (defined in **menutab.c)** and the value that indexes into the **tftp\_menu**’s item list, specifically, to the **GO** entry. This item entry contains a pointer to the function **fileio\_tftp()** (defined in **../uboot/lib\_ids/fileio.c**). **item\_input()** calls this function with an argument pointer to the **tftp\_menu.**

**fileio\_tftp** first sets the appropriate u-boot variables from the respective fields in the **tftp\_menu**. Note: prior to entry to the tftp download menu, a pointer to the selected TLS list menu (the one the user selected to download) was placed in the tftp download menu’s **menu** field. This function uses this menu to loop through the list of TLS files and download them.

For each file in the TLS file list, the TLS filename is loaded into the u-boot global variable **BootFile**. It then verifies that the number of TLS files already loaded does not exceed 14 (need at least 1 empty slot for the current TLS file about to be downloaded). It advances the load address for the current TLS file to the next available location in local memory, and puts the new load address into the u-boot environment variable **load\_addr**. Then the u-boot function **NetLoop()** is called with the argument **TFTP** which tells **NetLoop()** to perform the download from the tftp server to local memory at the specified load address. During the actual download, a progress bar, in the form of increasing ‘**#**’ symbols shows progress made during this individual download. Upon return from u-boot, the size of the file just loaded, in bytes, is saved in the TLS header’s **size** field in local memory and the results are displayed on the IDS screen. Following is an example of the results output:

**ids> 1#####################################539715:83C43**

**[ TEST LIST tcm/ppc.tls tls @ 10000000 539715 1 DOWNLOAD \*\*\*\* PASSED \*\*\*\* ]**

**#################247488:3C6C0**

**[ TEST LIST tcm/ram-lite.tls tls @ 10083C43 247488 2 DOWNLOAD \*\*\*\* PASSED \*\*\*\* ]**

**Hit any key to continue ...**

Each results display for a single downloaded TLS file shows in 2 lines. The progress bar and the downloaded file size, which is displayed in both decimal and hexadecimal (**539715:83C43)**, appear in the first line. The second line displays the TLS filename, the address to which it was downloaded (**tcm/ppc.tls tls @ 10000000)** and the size in decimal plus a **PASSED** (or **FAILED**) indication.

## Menu Build Process

The menu build process for standalone tests is a continuation of the download process. After the download process finishes and the user responds to the concluding message posted at the end of that process, IDS auto builds the menus needed to process the TLS files and IDS tests that were downloaded. It traverses the downloaded TLS files, with their associated IDS files, and builds menus in a separate menu memory space which are used by IDS to run tests.

Note that the starting address of the pool of memory allocated to menus is **0x00201000**, and the size of the pool is over 3 Mbytes. This number is hard coded into **menutab.c** in the **ram\_carve\_menu** which provides board-specific memory allocations for various IDS memory requirements. Also remember that the menu structure definition includes 16 item structures to hold the data appropriate to the structure type and that the first item is always preset to “**1) RUN**”. If the menu is a list type, then each entry is a TLS menu which points to a list of IDS tests; if the menu is a test type, then each entry points to a set of arguments for that test. For a complete list of the various types a menu item can be, see the types definitions near the beginning of the file **../uboot/include/ids/menu.h**.

## Structures

There are several structures that are common across menus and provide the support needed to execute the download process. The structures are **menu\_t**, **item\_t**, **idstime\_t, hwreg\_t** and **idsfs\_t**, and they are located in:

**../uboot/include/ids/menu.h**

**menu\_t** is the primary menu structure, and the others are in a supporting role, i.e., these subordinate structures are used in the **menu\_t** definition. The main structures, **menu\_t** and **item\_t** are described here. The others are simple enough to understand and can be found in the header file noted above.

## menu\_t

This structure defines the common menu format for all menu types: list menus, control menus and test menus. It’s located in the file **../uboot/include/ids/menu.h**.

**/\***

**\* This structure describes one menu within the IDS framework.**

**\*/**

**typedef struct menu**

**{**

**char name[32];**

**int itemcount;**

**// item\_t itemlist;**

**item\_t itemlist[16];**

**} menu\_t;**

The **name** field may contain up to 31 characters for a menu name. The **itemcount** and the **itemlist** fields are related: the count field shows how many **item\_t** elements are contained in this menu. For most menus, the first item in the **itemlist** is the “**1) RUN**” menu selection that, when selected, executes all tests defined at this menu level in the menu tree. Therefore, the maximum number of items in the **itemlist** is 15.

## item\_t

This structure defines the format for the **menu\_t** structure’s **itemlist** array of items such as TLS file lists, IDS test lists and IDS test arguments. The number of items in this array is fixed at 16. In those menus involved in control functions, such as the **tftp\_menu** or the **execution\_menu**, the first item field is the “**1) GO**” selection. For menus associated with test lists and tests, the first item field is the “**1) RUN**” selection. That leaves 15 item entries available for other menu-related data.

This structure is located in the file **../uboot/include/ids/menu.h**.

**/\***

**\* This structure describes one item within the list of items that**

**\* comprise a menu.**

**\*/**

**typedef struct item**

**{**

**char name[32]; /\* Item display name \*/**

**int type; /\* Item type \*/**

**struct menu \*menu;**

**void (\*func)(struct menu \*);**

**union**

**{**

**char cache[128]; /\* Item data \*/**

**idsfs\_t idsfs; /\* Overloaded: tls/ids ptrs \*/**

**hwreg\_t reg; /\* Register item data \*/**

**idstime\_t time; /\* IDS time \*/**

**} data;**

**char help[128]; /\* Help message \*/**

**// unsigned long eol; /\* 0x34567890 \*/**

**} item\_t;**

As before, the **name** field may contain up to 31 characters for an item name. The **type** field identifies what purpose the specific item has: it could be another menu, a TLS test list, an IDS test, an application, etc.

The **menu** field points to the menu associated with the **type** of item this is. For example, if the **type** is **ITEM\_APPL**, then the **menu** field is set to NULL, and the **(\*func)(struct menu \*)** field points to an appropriate executable function.

Another example applies if the **type** is **ITEM\_MENU**. The **menu** field is pointed to another menu, and the **(\*func)(struct menu \*)** field points to the **menu\_input()** function which processes menus.

The overloaded **data** field holds a variety of data: **cache** can contain the name of the TLS file to download, **idsfs** holds TLS and IDS header pointers which point to corresponding headers in download memory, etc.

Clearly this shows how flexible the structures are. They can be used for a variety of purposes and still work within the framework of the structure definitions. Here are two examples of how this overloaded **data** field can be used.

One example is provided by the build process where the default “**1) RUN**” entry is placed into an executable menu, both TLS and IDS.

In Builder.c in **../uboot/lib\_ids/**, the function “**menu\_insert\_run(menu\_t \*p, list\_hdr\_t \*tls, test\_hdr\_t \*ids)**” fills out the first entry in the 16 member **itemlist[16]** structure as the run entry. With **\*p** pointing to the current menu structure and **\*tls** and **\*ids** pointing to the target TLS menu and the first IDS menu in the TLS menu item list, respectively, the following shows how this overloaded field is set up to point to menus that will be parsed and executed by the test process:

**p->itemlist[0].data.idsfs.tls = tls;**

**p->itemlist[0].data.idsfs.ids = ids;**

A second example of how this overloaded **data** field is used is to describe a hardware register parameter, i.e., the structure **hwreg\_t**. This structure has 3 members: **addr** (32 bytes), **mask**(32 bytes) and **data**(32 bytes). It is used in an IDS test that references a hardware register. The structure members translate to 3 individual parameters. In the “**Makefile**” that defines the register based IDS test, the parameters would look like this:

**control = {0x80010000:0x8FFE) \**

**reg-data= (0x80010002:0x0000} \**

**.**

**.**

The string to the left of an ‘=’ sign is the register name of an IDS menu parameter, and the ‘**:**’ separated values within the braces, “**{…}**”, represent the register address and the read/write mask, in this case, a 4-byte value. The data portion of the **hwreg\_t** structure defaults in the menu build process to a string that represents a “0”. The way this is done in the menu build process in the code looks like this:

Given: **char \*cachep** is set to point into the opening brace (‘**{**‘) in the register menu parameter string, then the **hwreg\_t** **addr** field of the IDS menu structure is set thus:

**sprintf(itemp->data.reg.addr, "%s", &cachep[1]);**

Next, the **cachep** pointer is advanced in the menu parameter to the ‘**:**’ separator and the mask field is set thus:

**sprintf(itemp->data.reg.mask, "%s", &cachep[1]);**

Finally, the **hwreg\_t data** field is set thus:

**sprintf(itemp->data.reg.data, "%s", 0);**

## Creating and Modifying Test Parameters

Test parameters are the means used by the IDS build process (Makefile) to create different tests from the same test object file. As such, these parameters are first defined when a test source file is created. The initial parsing of the parameters from the argument list (argc/argv mechanism) passed into the test when it is invoked by the test execution engine sets the order in which the parameters must thereafter be placed. This order must be replicated exactly in the Makefile when new test instantiations are created.

An example of this parameter ordering and how it’s handled in a test is provided by the source file “**fpga-load.c**”. There are multiple variations supported by this test. The invocation of this test in Makefile should look like one of the following where the “**%s**” is the new test name and the rest are parameters. Note that the usage statements are taken directly from the source test file:

**Usage: %s [mux | hub | xrtr] location size**

**Usage: %s [mux | hub | xrtr] -stop channel**

**Usage: %s [mux | hub | xrtr] -start channel**

**Usage: %s [mux | hub | xrtr] -reset channel**

Some examples of how this is used in Makefile follow:

1. This example represents the first Usage statement, above:

**$(MKTEST) fpga-load -o hub-load fpga=hub \**

**memory=!loadaddr length=!filesize**

where:

%s new test name = **-o hub-load**

device accessed = **hub**

location in memory = **!loadaddr** which is the load address from u-boot

size of load image = **!filesize**  which is the size variable from u-boot

1. This second example shows the

**$(MKTEST) fpga-load -o hub-start fpga=hub op=-start chan=all**

where:

%s new test name = **-o** **hub-start**

device accessed = **hub**

op to perform = **-start**

target channel(s) = **all**

## Menutab.c

This file contains both the core IDS menus required to process standalone tests and a number of “list of lists” TLS menus that encapsulate sequences of individual TLS files and the IDS tests contained within. These serve as a starting point from the initial **Mode Menu** from which to execute the current crop of tests. All this is described in detail in the “**Integrated Diagnostics System (IDS) Functional Specification** in section 8.

## Core IDS Source Files

The core IDS source files are located in:

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

The following table provides a list and brief description of these files:

|  |  |  |
| --- | --- | --- |
| 1 | builder.c | Builds menus from downloaded TLS files |
| 2 | cmd.c | Support for IDS hotkey commands |
| 3 | exec.c | Executes the tests by walking the menus |
| 4 | fileio.c | Provides I/O support such as tftp |
| 5 | flash.c | Support for flash devices; not fully implemented |
| 6 | mailbox.c | Initial attempt to replace “jmpvec” table with mailbox mechanism; not fully implemented |
| 7 | main.c | Intializes IDS; entry point from u-boot |
| 8 | menu.c | menu processing support |
| 9 | menutab.c | IDS core menus; standalone top level menus |
| 10 | memtools.c | Peek and poke support |
| 11 | patgen.c | Pattern generator, including random |
| 12 | pgen.c | Packet generator support; not used in TCM |
| 13 | progress\_bar.c | Progress bar; not fully implemented |

## Walking the Menu Tree

Walking the menu tree is an user initiated activity that starts at the first menu encountered upon entering IDS. That is the **Main Menu**. The function called by IDS initialization is **menu\_input()**. This function is the main menu processing routine. The argument to this function is a menu pointer which this function translates into action. In the case of the **Main Menu**, the 3rd menu entry is “**3) IDS**” whose **menu** field (see **item\_t** structure above) points to the **Mode Menu**, the menu from which all other menus flow. Following is a description of the 2 primary menu walking functions used by IDS. Both functions are in the following u-boot file: **../uboot/lib\_ids/menu.c**.

**menu\_input()** is called with a pointer to a menu. It displays the menu and takes user input in the form of a selection of one of the items in the menu. Then **item\_input()** is called with 2 arguments: the same menu pointer and the user selected item from that menu in the form of an index into the menu’s item list. This function checks the **type** (see structure **item\_t**, above) field in the selected item to determine the next course of action. If the item type is a menu, then the function field of this item’s structure points back to **menu\_input()**. **item\_input()** calls through the function field to **menu\_input()** passing in as its argument the menu pointer contained in the item’s **menu** field, the next menu to process. This sequence of switching between menu and item processing continues until it stops at either the TFTP download menu or the test execution menu.

At any of the 3 menu levels, top level list menu, second level TLS list menu and third level IDS test menu, if the user selects the “**1) RUN**” entry, IDS will walk the menus, as described above, to get to the third level IDS tests to execute them. Read the following sections that describe the 3 menu levels as they are built, and also look at the memory layout of menus in section 5.6 to see a graphic representation of the menu levels.

Walking back up through the menus (towards the top level **Main Menu**) is also user initiated (by entering either the escape key or the letter ‘**u**’) and results in IDS unwinding the nested **menu\_input()** and **item\_input()** functions in the reverse order that they were called, one menu level at a time.

## Building Menus from TLS Files

The primary build function is **menu\_tls\_builder()** in **../uboot/ids\_lib/builder.c**. IDS calls the function **menu\_list\_appl()** passing in the address to the **Mode Menu** as an argument (which is not actually used in the called function).

The **menu\_list\_appl()** fetches a pointer to the first TLS file in download memory. It then calls **menu\_tls\_builder()** passing in as arguments the pointer to the menu **list\_menu** (base location where downloaded TLS and IDS menus are stored as they are built) and the pointer to the first TLS file in download memory.

The pointer to TLS file memory is actually pointing to the TLS header for the first downloaded TLS file. The first structure located at the menu **list\_menu** (**name** is **master-list MENU**) is the TLS header for that menu. This provides pointers to both the downloaded files and to the menus being built. Two local pointer variables are set as follows: **tlsp** points to the first TLS header in download memory, and **idsp** points to that TLS header’s first IDS header.

**menu\_tls\_builder()** starts by setting some global variables: **menu\_pool** points to the start of menu storage in local memory; **menu\_index** is set to 0; and the **list\_menu**’s itemcount (number of items added to the menu) is set to 0. Next, the first item in the item list is pre-set to “**RUN**” along with the other settings required for this item: **type** is set to **ITEM\_APPL**; **menu** points to **list\_menu**; the function field points to **test\_execution\_appl()** which, when invoked, brings up the **Execution Menu** that is the lead in to test execution; the tls and ids pointer fields point, respectively, to the TLS header in download memory and the IDS header for the first test header under that TLS header; and, finally, the **list\_menu**’s itemcount is set to 1.

At this point, IDS sets a loop to deal with all TLS level files in download memory. The loop starts with the **list\_menu**’s **itemcount** set to 1 by the previous processing. IDS stays in the loop as long as the **itemcount** doesn’t exceed the number of TLS files stored in the **entries** field of TLS entry in the **dir\_tbl[]** array of structures described in the **Structures** sub-section under **DOWNLOAD PROCESS**, above.

In this loop, **menu\_tls\_builder()** continues with setting the information from the TLS header in download memory into the second (**itemlist[1]**) item in **list\_menu**’s item list. This becomes the first actual TLS entry in the primary execution menu. It sets the item’s **itemcount** field to the count from the TLS header in download memory. This identifies how many IDS tests are associated with this TLS header. It creates an unique TLS name in this item’s **name** field by concatenating the TLS name in the TLS header from download memory with the item’s **itemcount** field in the form “**TLS name-itemcount**” and storing the resulting unique name here. It sets the item’s **type** field to **ITEM\_LIST**.

Next, IDS allocates a menu sized chunk of memory from the menu pool and assign it to the TLS menu. The local **menucount** variable is incremented to show that a menu slot has been used. For the duration of this function, this variable tracks the number of menus allocated. The **menu** field of this item is set to point to the allocated memory chunk, and the function field is set to **list\_input()** which, when invoked during test execution, will display the menu in the **menu** field and allow user input to determine which action will be taken next. Finally, the name field of the newly allocated menu is set to the **name** field in this item which was just set with an unique name (see above).

Next, **idsp**, is set to point to the first IDS header in the current TLS file in download memory. The **RUN** entry is initialized in the first item of the new TLS menu just allocated (see above for details). The **itemcount** field of the new TLS menu is set to the count contained in the TLS header in download memory. This count represents the number of IDS tests contained under this TLS entry.

IDS sets up a second nested loop to parse through the set if IDS files in download memory for each TLS entry. The **testindex** is set to 1 (**RUN** has been pre-set to the 0th list position). IDS stays in the loop as long as **testindex** doesn’t exceed the count of IDS tests in this TLS entry.

For each IDS test, the first step is to create an unique name for this test in a similar manner as was done for the TLS menu name: the IDS test name from the download IDS structure is concatenated with the TLS menu’s **itemcount** plus the current **testindex** in the form “**IDS test name-itemcount.testindex**”. This name is stored in the IDS test’s **name** field in the current TLS’s menu’s item slot referenced by **testindex**. The IDS test’s **type** field in the current TLS’s menu’s item slot is set to **ITEM\_TEST**.

Next, IDS allocates a menu sized chunk of memory from the menu pool and assigns it to the IDS test menu. The local **menucount** variable is incremented to show that another menu slot has been used. The IDS test’s **menu** field in the current TLS’s menu’s slot is set to point to the allocated memory chunk, and the function field is set to **menu\_input()** which, when invoked during test execution, will display the menu in the **menu** field and allow user input to determine which action will be taken next. Finally, the **name** field of the newly allocated menu is set to the IDS test’s **name** field in the current TLS’s menu’s slot which was just set with an unique name (see above).

The **RUN** entry is initialized as the first item of the new IDS menu just allocated (see above for details). The IDS menu’s item count, which represents parameters, or arguments, to the test, is set from the **argc** field in the IDS test header in download memory.

IDS sets up a third nested loop to parse through the set if parameters in download memory for each IDS entry. The **parmc** is set to 1 (**RUN** has been pre-set to the 0th list position). IDS stays in the loop as long as **parmc** doesn’t exceed the count of parameters in this IDS entry.

The **name** field of this parameter entry is set to the **name** field from the corresponding parameter entry in the corresponding IDS file in download memory. The **type** field is set to **ITEM\_PARM**. The **menu** field is set to **ids\_svc** (initially set to 0), and the **func** field is set to 0.

IDS then performs some parameter checks:

- if the parameter contains the string “access” and an access type is also provided (char, short, long, longlong), then the **access** variable is set accordingly. If the access type is missing, it defaults the variable to sizeof long.

- if the current parameter has the string “ids-svc”, then this is a service menu. Currently, only TFTP service is provided. The **type** field of this parameter is set to **ITEM\_SVC**. The **func** field is pointed to **fileio\_tftp\_svc()** located in **../uboot/lib\_ids/fileio.c**. Also, this menu’s **RUN** entry’s func field is also set to **fileio\_tftp\_svc()**. The local variable **ids\_svc** is set to point to the start of this IDS menu.

- if the current parameter is a register (delimited by open and close braces ‘{‘ and ‘}’) and an access mask is included in the parameter (delimited by a colon ‘:’), then the mask field of this parameter’s entry, <item>.**data.reg.mask**, is set according to the **access** variable described previously. Finally, the **type** field of this parameter is set to **ITEM\_REG**.

IDS returns to the top of the third loop.

At the IDS test menu loop, the pointer to the start of the current IDS header in download memory, **idsp**, is incremented to the next IDS header, and IDS returns to the top of the second loop.

At the IDS TLS menu loop, the pointer to the start of the current TLS header in download memory, **tlsp**, is incremented to the next TLS header, and IDS returns to the top of the primary loop.

## Menu Layout in Local Memory

This section describes the menu layout in memory after the Build Process has finished its work. In this description, assume that there are 2 TLS files downloaded with 2 IDS files in the first one and 2 IDS files in the second.

**Top Level master-list Menu Layout – Top Tier**

**tlsbase = start of TLS download memory**

**tlsp = pointer to TLS header in download memory**

**idsp = pointer to IDS header in download memory**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| name = | master-list |  |  |  |  |  |
| itemcount = | 3 |  |  |  |  |  |
| item 1 |  |  |  |  |  |  |
|  | name = | RUN |  |  |  |  |
|  | type = | ITEM\_APPL |  |  |  |  |
|  | \*menu = | &master-list |  |  |  |  |
|  | \*func = | test\_execution\_appl() | |  |  |  |
|  | data.idsfs.tls = | &tls\_1 hdr, tlsp = tlsbase[0] | |  |  |  |
|  | data.idsfs.ids = | &tls\_1[1], idsp = tlsp[1] | |  |  |  |
|  | \*help = | N/A |  |  |  |  |
| item 2 |  |  |  |  |  |  |
|  | name = | tls\_1-1 |  |  |  |  |
|  | type = | ITEM\_LIST |  |  |  |  |
|  | \*menu = | list\_level-1 alloc TLS list menu | | **TLS List Menu**  **next page** | | |
|  | \*func = | list\_input() |  |
|  | data.idsfs.tls = | &tls\_1 hdr, tlsp = tlsbase[0] | |
|  | data.idsfs.ids = | &tls\_1[1], idsp = tlsp[1] | |
|  | \*help = | N/A |  |
| item 3 |  |  |  |
|  | name = | tls\_2-2 |  |
|  | type = | ITEM\_LIST |  |
|  | \*menu = | list\_level-2 alloc TLS list menu | |
|  | \*func = | list\_input() |  |
|  | data.idsfs.tls = | &tls\_2 hdr, tlsp = tlsbase[1] | |
|  | data.idsfs.ids = | &tls\_2[1] , idsp = tlsp[1] | |
|  | \*help = | N/A |  |

**TLS List Menu Layout – Second Tier**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| name = | tls\_1-1 |  |  |  |  |  |
| itemcount = | 3 |  |  |  |  |  |
| item 1 |  |  |  |  |  |  |
|  | name = | RUN |  |  |  |  |
|  | type = | ITEM\_APPL |  |  |  |  |
|  | \*menu = | &master-list |  |  |  |  |
|  | \*func = | test\_execution\_appl() | |  |  |  |
|  | data.idsfs.tls = | &tls\_1 hdr, tlsp = tlsbase[0] | |  |  |  |
|  | data.idsfs.ids = | &tls\_1[1], idsp = tlsp[1] | |  |  |  |
|  | \*help = | N/A |  |  |  |  |
| item 2 |  |  |  |  |  |  |
|  | name = | ids\_1-1.1 |  |  |  |  |
|  | type = | ITEM\_TEST |  |  |  |  |
|  | \*menu = | test\_level-1 alloc IDS test menu | | **IDS Test Menu**  **next page** | | |
|  | \*func = | menu\_input() |  |
|  | data.idsfs.tls = | &tls\_1 hdr, tlsp = tlsbase[0] | |
|  | data.idsfs.ids = | &tls\_1[1], idsp = tlsp[1] | |
|  | \*help = | N/A |  |
| item 3 |  |  |  |
|  | name = | ids\_2-1.2 |  |
|  | type = | ITEM\_TEST |  |
|  | \*menu = | test\_level-2 alloc IDS test menu | |
|  | \*func = | menu\_input() |  |
|  | data.idsfs.tls = | &tls\_2 hdr, tlsp = tlsbase[0] | |  |  |  |
|  | data.idsfs.ids = | &tls\_2[1] , idsp = tlsp[2] | |  |  |  |
|  | \*help = | N/A |  |  |  |  |

**IDS Test Menu Layout – Third Tier**

|  |  |  |  |
| --- | --- | --- | --- |
| name = | ids\_1-1.1 |  |  |
| itemcount = | 4 |  |  |
| item 1 |  |  |  |
|  | name = | RUN |  |
|  | type = | ITEM\_APPL |  |
|  | \*menu = | &master-list |  |
|  | \*func = | test\_execution\_appl() | |
|  | data.idsfs.tls = | &tls\_1 hdr, tlsp = tlsbase[0] | |
|  | data.idsfs.ids = | &tls\_1[1], idsp = tlsp[1] | |
|  | \*help = | N/A |  |
| item 2 |  |  |  |
|  | name = | memory |  |
|  | type = | ITEM\_PARM |  |
|  | \*menu = | 0 |  |
|  | \*func = | 0 |  |
|  | data.cache | 0x00400000 |  |
|  | \*help = | N/A |  |
| item 3 |  |  |  |
|  | name = | len |  |
|  | type = | ITEM\_PARM |  |
|  | \*menu = | 0 |  |
|  | \*func = | 0 |  |
|  | data.cache | 0x100000 |  |
|  | \*help = | N/A |  |
| item 4 |  |  |  |
|  | name = | cycles |  |
|  | type = | ITEM\_PARM |  |
|  | \*menu = | 0 |  |
|  | \*func = | 0 |  |
|  | data.cache | 1 |  |
|  | \*help = | N/A |  |

## How Test Execution uses Menus

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.

Also note that all execution functions are in: **../uboot/ids\_lib/exec.c**

* 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 performs the test cycle counting 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. It’s also the point at which the test executable is verified as a valid ELF image and where the ELF image is loaded and booted. The core IDS file that does this is: ../uboot/lib\_ids/fileio.c.

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. **Core IDS Board specific issues**

The one file in Core IDS that is Board specific is menutab.c. Many, but not all, of the menus defined in this file are specific to the board upon which IDS is being run. Some of the menus are platform independent and are part of the IDS infrastructure. As such, they will never be removed.

Other menus that provide the top level menu control for testing are, of necessity, board specific. For now, there is no mechanism in IDS to make these menus standalone as has been done with the IDS tests. Until such time as IDS Core is altered to provide this support, these menus must be kept here.

## Hot Key Processing

IDS provides several “hot” keys for special functions. Those functions include displaying the **Ram Carve** menu that shows how IDS uses memory, handling “**PASS**” and “**FAIL**” policy user modifications and help messages (only in **DEBUG** mode).

To engage one of these functions, the user enters a **CTRL + <key>** combination at the “**ids>**“ prompt. Some of the hot keys are menu or mode specific while others can be entered at any menu. The user input function, **ui()**, located in the **menu.c** file first filters all non-hot key input before passing the hot key entered to the **list\_input\_hotkey ()**, also located in **menu.c**. This function filters through the possible valid hot key inputs and processes them.

To return from the hot key function to the menu at which it was invoked, the user enters either the **ESC** key or either the ‘**u**’ key or the ‘**U**’ key.

All hot key processing is managed from **../uboot/lib\_ids/menu.c**.

## Show “RAM CARVE” menu - ^T

This hot key can be entered at any menu level. When entered, IDS displays the **Ram Carve** menu which shows how IDS allocates memory for the various processes it performs:

1. U-BOOT 0x00200000
2. MENU POOL 0x00201000
3. LOG FILES 0x00300000
4. RAM 0x00400000
5. TLS FILES 0x10000000
6. SYSTEM 0x17000000

Although it’s not recommended, the user can actually change any of these allocations by entering the index number of the item desired. IDS presents the full parameter at the ids prompt, puts the cursor at the beginning of the hex allocation position and allows the user to modify it. Entering the **RETURN** key sets the new value to the corresponding parameter location.

## Modify “PASS” policy - ^P

This hot key is only valid for a second level TLS menu that contains IDS tests. This menu includes columns for **PASS** and **FAIL** policies (see the policies which are previously described in this document). When the user enters this hot key, IDS calls the function **menu\_pass()** located in the file **../uboot/ids\_lib/menu.c** passing in the pointer to the current menu.

The menu, with the default policies, is left on the screen. IDS changes the prompt from “**ids>**” to “**pass>**”. The user selects a specific polity to change by entering the index for the one desired. IDS presents that policy from the selected parameter in the form:

**pass> <index> PASS = <policy>**

IDS then puts the cursor at the start of the policy position and allows the user to modify it. The user can enter any of the policies described previously in this document. Entering the **RETURN** key sets the new policy to the corresponding parameter location.

When the user exits this **PASS** mode by using the **ESC** or **u/U** keys, IDS redisplays the “**ids>**” prompt.

## Modify “FAIL” policy - ^F

This hot key is only valid for second level TLS menu that contains IDS tests. This menu includes columns for **PASS** and **FAIL** policies (see the policies which are previously described in this document). When the user enters this hot key, IDS calls the function **menu\_fail()** located in the file **../uboot/ids\_lib/menu.c** passing in the pointer to the current menu.

The menu, with the default policies, is left on the screen. IDS changes the prompt from “**ids>**” to “**fail>**”. The user selects a specific polity to change by entering the index for the one desired. IDS presents that policy from the selected parameter in the form:

**fail> <index> FAIL = <policy>**

**IDS then puts the cursor at the start of the policy position and allows the user to modify it. The user can enter any of the policies described previously in this document. Entering the** RETURN **key sets the new policy to the corresponding parameter location.**

When the user exits this **FAIL** mode by using the **ESC** or **u/U** keys, IDS redisplays the “**ids>**” prompt.

## Menu HELP - ^N

This hot key is only valid in **DEBUG** mode (selected from the **Mode Menu**). This is the mode that handles the built in tests. Since they are part of the IDS core build, they actually have help messages. Help support is not provided in IDS for standalone tests at this time.

This hot key can be entered at any **DEBUG** mode menu level. When the user enters this hot key, IDS calls the function **menu\_help()** located in the file **../uboot/ids\_lib/menu.c** passing in the pointer to the current menu.

The menu is left on the screen. IDS changes the prompt from “**ids>**” to “**help>**”. The user selects an item from the menu by entering the index for the one desired. At the **help** prompt, IDS presents the selected parameter along with the associated help message. For example, given the following menu:

**1) NAND**

**2) RAM**

**help>**

if the user selects “**1) NAND**”, IDS presents the following at the **help** prompt:

**help> 1 NAND Test menu**

Entering the **RETURN** key clears the help message from the prompt leaving just the **help>** prompt waiting for the next help request.

When the user exits this **HELP** mode by using the **ESC** or **u/U** keys, IDS redisplays the “**ids>**” prompt.

## TLS Download Control - ^L

This hot key is only valid when used when the “**file\_mgr\_menu**” (see in **menutab.c**) is being displayed. In the **Mode Menu**, only entries “**4) INSTALLED TESTS**” and “**5) BRINGUP DVT**” are tied to that menu through their **item\_t** structure’s **func()** pointer which points to the function **config\_regress\_appl()** located in **../uboot/ids\_lib/menutab.c**. That function, which is called when either of those entries are selected from the **Mode Menu**, calls the **menu\_input()** function passing in a pointer to the “**file\_mgr\_menu**”.

That function displays the menu, a second tier TLS menu, within which this hot key can only be used. Interestingly, selecting the “**1) RUN**” entry in this menu also has the same result. Either action brings up the TLS Download Menu through which tftp downloads are controlled.

The actual filtering that checks for that specific menu and processes this hot key is the function **ui()** located in **../uboot/ids\_lib/menu.c**. This function is called from the **menu\_input()** function, in the same file, which processes menus.

## IDS Commands Processing

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.

The user invokes these commands from the “**ids>**” prompt at any menu by entering the desired command at the prompt. This entry is processed by the function **ui()** located in **../uboot/ids\_lib/menu.c**.

After checking the keyboard input for all other conditions, IDS checks the first character in the input to confirm it’s in the printable text range of the ASCII character set, between space (0x20) and ~(0x7E). If the check is positive, then IDS calls the function **cli()**, located in the file **../uboot/lib\_ids/cmd.c,** passing in a pointer to the start of the keyboard input.

The **cli()** function confirms that a valid command has been entered. All commands, except **help**, require arguments. IDS parses the arguments into an **argv/argc** construct and calls the requested command function passing the **argv** and **argc** arguments. Upon command completion, IDS presents a **PASSED** or **FAILED** message, depending on the command’s return status, and pauses for the user to enter any key.

The functions implementing the commands are located in different files and directories. The list of commands and related files is located in the “**Integrated Diagnostics System Functional Specification**” and can be viewed there.