

## **Programmers' Guide to WV5 C-Script**

Revision: 0.9

Date: May 5, 2009

### **Revision History**

Rev. 0.9    May 5, 2009

Document corresponds with the P1R2 release of the WaveVision-5 package.

## Table of Contents

<b>1</b>	<b>REFERENCES.....</b>	<b>5</b>
<b>2</b>	<b>COMPLETE DOCUMENT SET.....</b>	<b>6</b>
<b>3</b>	<b>INTRODUCTION.....</b>	<b>7</b>
<b>4</b>	<b>TERMINOLOGY .....</b>	<b>8</b>
4.1	GENERAL CODING CONVENTION USED IN THE SCRIPT .....	8
<b>5</b>	<b>GENERAL DESCRIPTION .....</b>	<b>9</b>
5.1	GENERAL LAYOUT .....	9
5.2	NAMES USED IN THE SCRIPT .....	9
5.3	GENERAL COMMENT REGARDING THE MANDATORY FUNCTIONS.....	9
<b>6</b>	<b>CODE DESCRIPTION BY SECTIONS.....</b>	<b>11</b>
6.1	CONSTANTS.....	11
6.2	REGISTER DEFINITION .....	11
6.3	DUT-SPECIFIC DATA STRUCTURE .....	12
6.4	STANDARD OBJECTS .....	13
6.5	CAPTURE RELATED STRUCTURE DECLARATIONS.....	13
6.6	REGISTERS DECLARATIONS.....	13
6.7	GUI RELATED DECLARATIONS .....	14
<b>7</b>	<b>FUNCTION DESCRIPTIONS .....</b>	<b>15</b>
7.1	MANDATORY BOARD FUNCTIONS - GENERAL, VERSION.....	15
7.1.1	NAMING .....	15
7.1.2	RETURN CODE .....	15
7.1.3	DEBUG OUTPUT .....	15
7.1.4	SETTING THE RETURN VALUE FOR THE DLL.....	16
7.1.5	SCRIPT FUNCTION RETURN .....	16
7.1.6	FORWARD DECLARATIONS FOR INITIALIZATION .....	16
7.2	MANDATORY DUT FUNCTIONS - GENERAL, CREATE.....	17
7.2.1	ADC14DS105_INIT_CAP_CONFIGS .....	18
7.2.2	ADC14DS105_INIT_REGISTERS .....	18
7.2.3	ADC14DS105_INIT_TABS.....	19
7.2.4	ADC14DS105_INIT_TAB0.....	19
7.2.5	Initializing the GUI tab.....	19
7.2.6	Initializing the entire control definition array.....	19
7.2.7	Initializing a combo box .....	19
7.2.8	Initializing a numeric edit.....	21
7.2.9	ADC14DS105_TAB0_REFRESH.....	21
7.2.10	Refreshing a combo box.....	22
7.2.11	Refreshing a combo box for cheaters.....	23
7.2.12	Refreshing a numeric edit .....	23
7.2.13	Sending a change notification.....	23
7.2.14	ADC14DS105_INIT_TAB1.....	24
7.2.15	Initializing a Level 1 Register definition.....	24
7.2.16	ADC14DS105_TAB1_REFRESH .....	25
7.2.17	Refreshing a level 1 register .....	25
7.3	MANDATORY DUT FUNCTIONS - GENERAL, DESTROY .....	26
7.4	MANDATORY DUT FUNCTIONS - GENERAL, EXECUTE_DUT_INITIALIZATION.....	26
7.5	MANDATORY DUT FUNCTIONS - GENERAL, GET_CLOCK_FREQUENCY .....	27
7.6	MANDATORY DUT FUNCTIONS - GENERAL, POLL_HOOK .....	27
7.7	MANDATORY DUT FUNCTIONS - GUI, GUI_TAB_ENUM .....	27

7.8	MANDATORY DUT FUNCTIONS - GUI, GUI_CONTROL_ENUM .....	27
7.9	FORWARD DECLARATIONS FOR SUPPORTING GUI CHANGES .....	28
7.10	MANDATORY DUT FUNCTIONS - GUI, GUI_CONTROL_CHANGE.....	28
7.10.1	ADC14DS105_TAB0_CHANGE.....	29
7.10.2	Changes on elements other than level 1 registers.....	29
7.10.3	ADC14DS105_TAB1_CHANGE.....	30
7.10.4	Changes on level 1 elements .....	30
7.11	MANDATORY DUT FUNCTIONS - CAPTURE, GET_CAPTURE_PARAMETERS.....	31
7.12	MANDATORY DUT FUNCTIONS - CAPTURE, EXECUTE_CAPTURE_ACTIONS.....	31
7.13	MANDATORY DUT FUNCTIONS - CAPTURE, EXECUTE_POST_CAPTURE_ACTIONS .....	31
7.14	MANDATORY DUT FUNCTIONS - DAC DATA DOWNLOAD PARAMETERS .....	31
7.15	MANDATORY DUT FUNCTIONS - DAC DOWNLOAD COMMAND PROCESS .....	31
7.16	MANDATORY DUT FUNCTIONS - GENERAL ACCESS, GENERAL_ACCESS_RD .....	32
7.17	MANDATORY DUT FUNCTIONS - GENERAL ACCESS, GENERAL_ACCESS_WR .....	32
<b>8</b>	<b>TYPICAL CUSTOMIZATION ISSUES .....</b>	<b>33</b>
8.1	CUSTOMIZING THE DUT HARDWARE INITIALIZATION .....	33
8.2	MODIFYING FREQUENCY CALCULATION .....	33
8.3	ADDING A TAB.....	34
8.4	ADDING A CONTROL TO AN EXISTING TAB .....	37
<b>9</b>	<b>SOFT SERIAL PROTOCOL.....</b>	<b>39</b>
9.1	OVERVIEW .....	39
9.2	SERIAL PROTOCOL ARRAY (WRITE ARRAY) .....	40
9.3	FIRMWARE RETURN ARRAY .....	40
9.4	EXAMPLES.....	41
9.4.1	EXAMPLE 1.....	41
9.4.1	EXAMPLE 2.....	42
9.5	SOFTWARE – GENERAL FLOW OF THE CODE .....	43
9.6	SOFTWARE SPECIFICS - DLL .....	44
9.6.1	IMAGE_MAP.XML.....	44
9.6.2	CHOOSING A DUT .....	45
9.7	SOFTWARE – SCRIPT.....	45
9.7.1	SENSORPATH2008.BRD.CPP .....	46
9.7.2	DETAILS OF CUSTOM SERIAL COMMUNICATION IN SENSORPATH2008.BRD.CPP .....	46
9.7.3	DETAILS OF THE ACTUAL PROTOCOL .....	47
9.7.4	DETAILS OF ROUTINE THAT SENDS THE COMMAND AND PROCESSES THE FIRMWARE RETURN ARRAY.....	47
9.7.5	USE OF SENSORPATH2008.BRD.CPP BY OTHER SCRIPT FILES .....	49
9.8	SENSORPATH2008_NONE_OPTICAL_ENCODER.DUT.CPP .....	50
9.9	SENSORPATH2008_NONE_SP1202S01RB.DUT.CPP .....	51
9.10	SENSORPATH2008_NONE_SP1602S01RB.DUT.CPP .....	51
9.11	CUSTOMIZING A SERIAL PROTOCOL ARRAY .....	51
<b>10</b>	<b>DEBUGGING.....</b>	<b>52</b>
10.1	CINT_TEST.EXE.....	52
10.1.1	LOCATION .....	52
10.1.2	RUNNING .....	52
10.1.3	COMMAND LINE OPTIONS .....	53
10.1.4	TESTING THE SCRIPT .....	53
10.1.5	FILES GENERATED .....	54
10.1.6	CRASH .....	55
10.1.7	JUNK IN DIAG.TXT.....	56
10.1.8	FAILURES TO IGNORE.....	57
10.2	USING A SIMULATOR FOR THE WV5 GUI .....	57
10.2.1	SIMULATOR LIMITATIONS .....	57

---

10.2.2	DATA\LogFiles\DLLLOG.TXT.....	58
10.2.3	EXETRACE.TXT .....	58
10.3	USING A REAL BOARD WITH THE WV5 GUI .....	58

# 1 References

*a) Introduction to the WaveVision-5 System* - high-level introduction: "what is it?"; "what can it do for me?"

*b) WaveVision5 Software Users' Guide.*

*c) WaveVision-5.1 Capture Board Users' Guide.*

*d) WV5 System Developers' Guide* - the first document one should refer to when developing hardware or software for the WV5 platform. Introductory material regarding the Script feature of WV5 is in this document.

*e) Programmers' Guide to WV5\_DLL API* - Detailed reference for those writing an application to run above WV5 Core software.

## 2 Complete Document Set

Following additional technical resources are provided with this document:

a) *WV5 C-Script Technical Reference* -

b) *Source header files:*

- `wvdl1.h` - public functions
- `wvdl1_defs.h` - public structures
- `wvdl1_private_types.h` - structures visible only to the DLL and the scripts.
- `script_def.h` - public structures of the C-Script
- `script_dll_interface.h` - interface between script and the DLL
- `script_api.h` - DUT API

c) *Example script file used:*

- `hardware\scripts\wv5_xc4vlx25_adc14ds105.dut.cpp`

In the process of developing a script file, you will need to work with the following log files:

- `diag.txt`
- `exetrace.txt`
- `Data\LogFiles\DLLLog.txt`

### **3 Introduction**

Scripting is a powerful feature of the WV5 Core software. With it the developer can customize the exact makeup of the GUI application running above the WV5 Core and also control precisely what hardware related actions will be taken by the WV5 Core software when the user issues a command from the GUI..

First description of the scripting function is provided in the WV5 System Developers' Guide. It is assumed that the reader has read that document before approaching this guide.

Please note that the WV5 Core software has also employed an XML-based scripting function in the past. This guide pertains to the new C-based scripting utility.

## 4 Terminology

Control definition

C structure that describes a GUI element

GUI element

Visible object in the register panel the user can interact with in the WV5 GUI

### 4.1 *General Coding Convention Used in the Script*

No tabs

Indents always 4 spaces

English "words" always separated by an underscore

Constants (#define)

Always all caps

Function names

Always all lowercase

Structure/enumeration type names

First letter of word always capitalized



## 5 General Description

### 5.1 General Layout

The script can be split into several large blocks. In the script, each of these blocks are enclosed in `BEGIN` and `END` comments. The example script used for the purposes of this guide (`hardware\scripts\wv5_xc4vlx25_adc14ds105.dut.cpp`) is of the following structure, which is typical of all scripts:

The first block contains all the constants.

The second block contains all the constants to define registers.

The third block contains the DUT-specific data structure.

The fourth and final block contains all the functions.

The functions can be grouped into two categories.

The first category contains the mandatory functions. These are the functions that all DUT scripts are required to implement. These functions are clearly labeled with the "MANDATORY DUT FUNCTIONS" in both this document and the script. The mandatory functions require a very specific naming convention. Please see Section 5 of `cscript.doc` for more details.

The second category contains all the functions that help assist the mandatory functions. Although there is not strict naming convention for these, they usually have the DUT name prepended to make sure they are unique to the script.

This document will mirror the structure in the script. The information for each of the major blocks will be contained in a separate chapter. Each mandatory function will be contained in a separate chapter. All the helper functions will be listed as a sub-item of the corresponding mandatory function.

### 5.2 Names Used in the Script

Since the DLL supports multiple DUTs and multiple boards simultaneously, all the functions and structure names must be unique so that multiple script files can be loaded at the same time without naming conflicts.

This is the reason most of the structures and constants are prepended with the DUT name. This is to make sure all the constants and structures used are only available in this script. This will guarantee no naming conflicts.

### 5.3 General Comment Regarding the Mandatory Functions

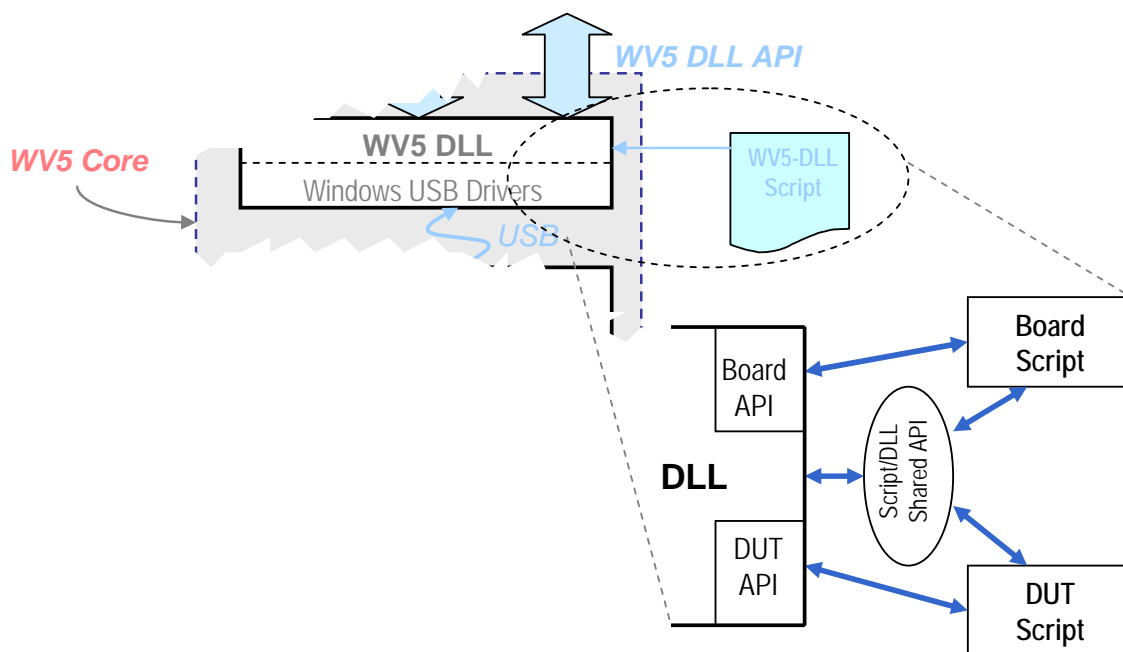
Each mandatory function requires a very specific name. The technical documentation for the APIs describes the names. Also the `cscript.doc` contains some information.

Most of these functions have only one parameter and that is a pointer to the DUT-specific data structure. All state pertaining to the DUT should be stored in this structure. The state should not be global.

Most of these functions have parameters and return values. These are passed indirectly. The parameters into the functions must be retrieved from the shared API. The return values must be set using the shared API.

The shared API is described in further detail in the WV5 System developers' Guide. The reader is reminded to read that overall guide before going further into this document.

The image below is taken from the WV5 System Developers' Guide and describes the general relationship of the DLL, API, and script.



## 6 Code Description by Sections

### 6.1 Constants

This section contains the constants that are used throughout the script.

```
#define ADC14DS105_VERSION "0.0.0.1"
```

This is the script version number that will be shown in the WV5 GUI's "Help->About" dialog box.

The general format is "major.minor.release.build". Whenever the script is changed, it's a good idea to increment one or all of these numbers.

```
#define ADC14DS105_NUM_SIMULTANEOUS_CH_CAPS 2
```

This is the number of simultaneous channels the FPGA can capture. This will be used later when initializing the capture configuration structures.

```
#define ADC14DS105_NUM_CAP_CONFIGS 3
```

This is the number of capture configurations this DUT will support. A capture configuration is a structure that contains all the information regarding how to capture data from the ADC.

The number describes the number of elements in the pull-down menu in the "Signal Source" tab of WV5. There will be more on the capture configuration later.

```
#define ADC14DS105_NUM_TABS 2
```

This is the number of register tabs in the GUI.

```
#define ADC14DS105_TAB0_NUM_DEFS 9
```

This is the number of control definitions in tab 0. Each tab should have its own constant for the number of definitions. These GUI constants will be reused heavily later in the script.

```
#define ADC14DS105_TAB1_NUM_DEFS 1
```

```
#define ADC14DS105_MAX_DEFS ADC14DS105_TAB0_NUM_DEFS
```

This constant should contain the maximum number of definitions for all the tabs. This constant is used to create an array that will help in refreshing the GUI.

### 6.2 Register Definition

The constants in this section should describe every field of every register that the user will manipulate. The listing should include all registers including those of the DUT and FPGA and other peripheral devices like a PLL.

There are two reasons for listing the registers in this manner. First is to centralize all the register information in one location. A change to a value here will be properly propagated to all the code that uses the constant.

The second is to simply give all the fields of all the registers a readable name and not just some numbers.

Each constant should have a prefix specific to the DUT and the register. The register list should consist of at least the following two fields:

```
#define ADC14DS105_DEV_CTRL_ADDR 0x0
```

This is the address of the register.

```
#define ADC14DS105_DEV_CTRL_DEFAULT 0x8
```

This is the default value of the register. This should be the default value desired by the author once the hardware boots up. This does not necessarily have to be the hardware default after power on reset. When the hardware is initialized, these `_DEFAULT` values will be used when writing the initial values into the device.

Each field of the register should have the following two entries:

```
#define ADC14DS105_DEV_CTRL_OM_P 6
#define ADC14DS105_DEV_CTRL_OM_S 2
```

`_OM` is the name of the field. In this case, OM is meant to stand for "Operational Mode".

The `_P` and `_S` stand for "position" and "size" respectively and describe the bit position and number of bits for the field. This is a convention used to identify the bit fields. The position is relative to the LSB. So a position of 0 is the LSB of the register.

These two constants are used as inputs into utility functions that help with the repetitive task of extracting values out of a field in an integer and depositing values into an integer.

### 6.3 *DUT-specific data structure*

This structure encapsulates all the data regarding the DUT. All the DUT data must be in a structure. Global variables are strongly discouraged.

The reason is the user may want two or more of the same DUT plugged into the PC at the same time. For instance, the user may want to be able to compare the performance of two DUTs without having to use two PCs.

In a situation like this, each DUT must have its own state. By encasing all the DUT data into a structure, multiple copies of the same structure can be used, each copy representing and controlling exactly one DUT.

If global variables were used, then the two DUTs would end up conflicting over the use of the global variable. This is an undesirable situation.

The process of managing the copies is done by the DLL and not the script. The script's job is to operate only one copy or instance of the data at any one time.

The script does have to create the copy, though. This is described later.

## 6.4 *Standard objects*

```
Wv5_Brd* board_ptr;
```

This is a pointer to a `Wv5_Brd` object. Just like each DUT has a DUT-specific structure, so do boards.

The idea is that all DUTs that connect to the same board will most likely share some of the same routines, like accessing the Cypress memory. So, instead of cutting-and-pasting the same code over and over again into all the different DUT scripts, the shared code is placed in the board script once. And then the DUT scripts can reference the board script.

And just like the situation of having two of the same DUT connected, there can be two or more of the same board connected. For this reason, each board has its own board-specific data structure.

```
const Dut_Eeprom* dut_eeprom;
```

This structure represents the DUT EEPROM's contents. The DUT EEPROM contains information regarding the name of the DUT and other operating parameters. The DLL uses the DUT EEPROM extensively, but the script will most likely not need to.

## 6.5 *Capture Related Structure Declarations*

This section contains the two arrays that describe how to acquire data from the ADC. This document will not describe in great detail every field of these two structures. Assuming a WV5 capture board compatible DUT, these structures will be initialized by calling the board script.

```
Wv_Capture_Configuration cap_config[ADC14DS105_NUM_CAP_CONFIGS];
```

This structure is defined in `wvdll_def.h`. This information contains all the information required by the WV5 GUI to properly display the options available for this capture. The basic information includes the name, number of bits, min/max capture sizes.

```
Wv_Priv_Cap_Config priv_cap_config[ADC14DS105_NUM_CAP_CONFIGS];
```

This structure is defined in `wvdll_priv_types.h`. This structure contains all the nitty-gritty on how the DLL acquires and de-interleaves the data from the ADC. Please consult both the header file and `capture.doc` for more information.

## 6.6 *Registers Declarations*

This section contains an integer for each register that was defined in the register constants section.

When the user interacts with the GUI and modifies a register, the script should update the value in this data structure first, and then write that modification out to the hardware.

The idea is for the script to always contain a copy of the most recent hardware register values so that the hardware and the script can remain in sync.

```
WvWord dev_ctrl;
```

This will contain the most recent value of the Device Control register.

## 6.7 *GUI Related Declarations*

This section contains the structures to support the GUI. These structures are defined in `wvdl_def.h`.

```
Wv_Gui_Tab tabs[ADC14DS105_NUM_TABS];
```

These are little structures that contain some basic information about the tab, like the tab name that is seen written sideways in the register panel of the WV5 GUI.

```
WvControlDefinition tab0_def[ADC14DS105_TAB0_NUM_DEFS];
```

This array of `WvControlDefinition` contains all the definitions to describe all the objects that are seen in `tab0`.

```
WvControlDefinition tab1_def[ADC14DS105_TAB1_NUM_DEFS];
```

This array of `WvControlDefinition` contains all the definitions to describe all the objects that are seen in `tab1`.

```
bool changed[ADC14DS105_MAX_DEFS];
```

This is a helper array that keeps track of which controls have changed on a tab. The reason for this array will be described a little later.

## 7 FUNCTION DESCRIPTIONS

### 7.1 MANDATORY BOARD FUNCTIONS - General, version

```
Interface_Ret wv5_xc4vlx25_adc14ds105_version() {
    script_debug_out_pre("Version:%s", ADC14DS105_VERSION);
    script_debug_out_endl();

    return script_process_return(
        script_set_return_version(ADC14DS105_VERSION),
        "wv5_xc4vlx25_adc14ds105_version"
    );
}
```

This returns the version number to the DLL. Although small, this function highlights many concepts that are used throughout the script.

#### 7.1.1 Naming

Please note the naming of the function. Basically, it is the script file name with the .dut.cpp stripped, and then appended with `_version`. This process of stripping the file name extension and appending the function name is used for all mandatory function.

#### 7.1.2 Return code

All mandatory functions, with the exception of the create function, return `Interface_Ret`. The DLL will always check the return code for success. The other functions in the script may or may not check the return code, but a function should always return at least either a success or fail condition.

As a rule all functions in the script should return some type of code. This helps in debugging.

#### 7.1.3 Debug output

There is very limited debug capabilities in the scripting environment. The easiest and most effective way to debug is to scatter debug output lines throughout the code. There are several helper functions to that can assist in debug out. These are defined in the shared API.

```
script_debug_out_pre("Version:%s", ADC14DS105_VERSION);
```

The format of this function is very similar to `printf`. The `_pre` indicator is just a hint saying that when the DLL outputs this to the log, the DLL will prepend the current time to the output.

```
script_debug_out_endl();
```

`_endl` is just a hint saying that this simply outputs a carriage return. All of the debug output helpers in the shared API will not print out a carriage return at the end. The caller must either explicitly put a carriage return at the end of the string or call the `_endl` function.

For example:

```
script_debug_out_pre("Version:%s", ADC14DS105_VERSION);
script_debug_out_endl();
```

and

```
script_debug_out_pre("Version:%s\n", ADC14DS105_VERSION);
```

Both of these will result in the same output in the log.

The reason for the explicit carriage return function is so that the caller has complete control over the formatting of the debug output.

#### 7.1.4 *Setting the return value for the DLL*

```
script_set_return_version(ADC14DS105_VERSION)
```

The only method to pass values between the DLL and the script is to use the shared API. The shared API will allow the script to set return values and allow the script to retrieve function parameters.

The reason these function parameters and return values are not explicitly passed in the function arguments is due to the architecture of the scripting environment. Basically, all of these script functions are dynamically created. Since the DLL cannot know which functions will exist, it cannot have code that calls these non-existent functions.

#### 7.1.5 *Script Function return*

```
return script_process_return(
    script_set_return_version(ADC14DS105_VERSION),
    "wv5_xc4v1x25_adc14ds105_version"
);
```

This is the preferred method of returning from a function. The `script_process_return` is technically just a pass-through type of function; it simply passes the first parameter as the return code.

However, `script_process_return` also prints to the log the time of the function call and the return code and anything passed in the second parameter. This can be used as a shortcut instead of explicitly calling several of the script debug output helper calls.

#### 7.1.6 *Forward declarations for initialization*

The initialization of the DUT-specific data structure requires several steps. Instead of placing all these steps into one huge function, several smaller functions are defined to initialize only a part of the DUT-specific data structure.

The functions themselves are not implemented here; only the names are declared. The reason for this is that I find it easier to follow the flow of the code with a top-down approach to function calls. These names are a hint of what's coming. But they will be implemented after the calls are used.

```
Interface_Ret adc14ds105_init_cap_configs(Adc14ds105_Data* data);
```

Initialize the capture configurations

```
Interface_Ret adc14ds105_init_registers(Adc14ds105_Data* data);
```



Initialize the default values of the registers. However, this function does not modify the hardware. The steps the manipulate the hardware occur later.

At this point in the script initialization, there is no guarantee the hardware is ready to be manipulated.

```
Interface_Ret adc14ds105_init_tabs(Adc14ds105_Data* data);
```

Initialize the tabs.

```
Interface_Ret adc14ds105_init_tab0(Adc14ds105_Data* data);
Interface_Ret adc14ds105_init_tab1(Adc14ds105_Data* data);
```

Initialize individual tabs.

```
Interface_Ret adc14ds105_tab0_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
);

Interface_Ret adc14ds105_tab1_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
);
```

These two functions process the state of the hardware and make sure all of the GUI controls are up to date. The term `_refresh` is to indicate that the GUI control values are to be refreshed from the current state of the hardware.

These will be described in further detail in the implementation.

## 7.2 MANDATORY DUT FUNCTIONS - General, create

```
Adc14ds105_Data* wv5_xc4vlx25_adc14ds105_create(Wv5_Brd* board_ptr)
```

This function creates a new instance of the DUT-specific data structure and then initializes the data.

```
Adc14ds105_Data* data = new Adc14ds105_Data;
```

The `new` keyword is a C++ keyword. The interpreter is a C++ interpreter. `new` is used instead of `malloc` because that is what the the interpreter documentation suggests.

There is one additional C++ specific keyword used and that is `bool`. For simplicity, `bool` can be considered an unsigned 8 bit character.

These are the only two C++ specific constructs used. Everything else is C compliant.

```
script_debug_out_pre("wv5_xc4vlx25_adc14ds105_create");
script_debug_out_endl();
```

Debugging output.

```
script_load_file("wv5_generic.dut.cpp");
```

As the comment in the code suggests, this line loads an additional script file for use. This script file contains a helper function that will assist in initializing the capture configurations.

```
script_get_params_dut_create(&(data->dut_eeeprom), NULL);
```

This is an example of how to retrieve function parameters from the shared API. In this case, the code is requesting that the pointer `data->dut_eeeprom` be initialized.

```
adc14ds105_init_cap_configs(data);
adc14ds105_init_registers(data);
adc14ds105_init_tabs(data);
```

These are calls to initialize the various structures within the DUT-specific data.

```
script_debug_out_pre("wv5_xc4vlx25_adc14ds105_create: done");
script_debug_out_endl();
```

More debug output.

```
return data;
```

Finally, the newly created and initialized data structure is returned. This function is the only function in the mandatory set that will return something other than `Interface_Ret`.

### 7.2.1 *adc14ds105\_init\_cap\_configs*

This section initializes the capture configuration structures of the DUT-specific data. Since this is a WV5 compatible DUT, all the code for the initialization has already been written. The script only has to call the init function.

```
ret = wv5_configure_cap_configs(
    ADC14DS105_NUM_SIMULTANEOUS_CH_CAPS,
    ADC14DS105_NUM_CAP_CONFIGS,
    data->cap_config,
    data->priv_cap_config
);
```

`wv5_configure_cap_configs` is the function that initializes the configuration. The required parameters are the number of channels the FPGA can simultaneously capture, the number of available configurations, and the configuration themselves.

For all the DUTs that use `wv5_configure_cap_configs`, the number of configurations required depends on the number of channels. If the FPGA only supports one channel, then only one capture configuration is necessary. If the FPGA supports two channels of simultaneous capture, then three configurations are necessary. The three configurations refer to capturing the A channel, capturing the B channel, and capturing both A and B simultaneously.

### 7.2.2 *adc14ds105\_init\_registers*

This copies the default register values into the DUT-specific data structure.

```
data->dev_ctrl = ADC14DS105_DEV_CTRL_DEFAULT;
data->usr_test_reg0 = ADC14DS105_USR_TEST_REG0_DEFAULT;
data->usr_test_reg1 = ADC14DS105_USR_TEST_REG1_DEFAULT;
```

These use the `_DEFAULT` constants defined in the register section.

### 7.2.3 *adc14ds105\_init\_tabs*

This function initializes all the tabs.

```
ret = adc14ds105_init_tab0(data);
if (ret != INTERFACE_RET_OK) {
    return ret;
}
```

These three lines of code are repeated for each tab.

### 7.2.4 *adc14ds105\_init\_tab0*

This initializes the structures for tab 0. The code in this function can be heavily copied and tweaked depending on the requirements.

```
WvWord num_defs = ADC14DS105_TAB0_NUM_DEFS;
WvControlDefinition* defs = data->tab0_def;
WvWord i;
```

Please read the code comment regarding these two lines. The idea is that within the body of the function, there are not specific references to tab 0. All the code will reference these three variables. By doing this, the body of the code can be copied into other scripts with no changes. This is valid as long as the new function also defines these two variables, but with changes to specify a different tab.

### 7.2.5 *Initializing the GUI tab*

The section here specifies the name and other attributes of the GUI tab.

```
data->tabs[0].caption = "Hardware Register Tab";
```

This is the name of the tab and will probably need modification all the time.

```
data->tabs[0].level = 2;
```

This describes the type of GUI elements allowed in the tab. Level 1 GUI elements should set `level` to 1. Level 2 GUI elements should set `level` to 2.

```
data->tabs[0].tags = "";
```

This is currently not supported. Please leave this field as "".

### 7.2.6 *Initializing the entire control definition array*

This call is mandatory and should not be removed.

```
script_gui_init_defs(num_defs, defs);
```

This is to make sure all the fields are set to a reasonable value (namely 0). This is necessary so that any fields that were accidentally not initialized will not cause any harm.

### 7.2.7 *Initializing a combo box*

A combo box is a pull down box. This is the most complicated GUI element supported because it can support a variable number of items in the pull-down menu.

If a combo box is not necessary, please skip ahead.

Note the use of `defs[i]`. These variables are not specific to tab 0. This means this entire block of code can be copied into another script with no further editing other than changing the number elements and the names of each element.

```
defs[i].Type = WvComboBox;
```

All elements have a type identifier. These are defined in `wvdl_def.h`

```
script_gui_combobox_create_list(&defs[i].ComboBox, 3);
```

To ease the initialization of the combo box, the shared API implements two functions that provide some assistance. The goal of these helper functions is to hide all of the memory management and implementation details necessary to allocate variable-length arrays.

The first function is to create the list of objects that will contain all the information.

`script_gui_combobox_create_list` does this. This needs to be called before any other list related functions.

The two parameters are the control definition and the number of items in the drop down box.

```
script_gui_combobox_add_item(&defs[i].ComboBox, 0, "Normal Operation (00)", 0);
```

The second of the helper functions is to add one item into the list. There are four parameters.

The first is the control definition. The second is the item location; location 0 is at the top of the drop-down list and higher the index, the lower in the list.

The last is the value assigned for that location. This value is typically the value that will be used to modify the register. In this case, when the user selects "Normal Operation", the value will be 0.

The value does not have to match the index.

```
script_gui_combobox_add_item(&defs[i].ComboBox, 1, "Default Test Ouput Mode (01)", 1);
script_gui_combobox_add_item(&defs[i].ComboBox, 2, "User Test Output Mode (10)", 2);
```

```
defs[i].ComboBox.ItemSelected = 0;
```

Initialize the current selection to 0.

```
script_gui_allocate_string(&defs[i].ComboBox.Caption, "Operational Mode");
```

This is an example of how to allocate a string for the control definitions.

`script_gui_allocate_string` must always be used when dealing with strings in control definitions. This function hides all of the behind-the-scenes memory allocation required to handle variable-length strings.

The main reason this is necessary is because these string can change size during the course of the script. If the string does change and the string needs to be expanded, then more memory needs to be allocated for the string. All of this memory management is handled by the shared API with the use of these helper functions.

Most of the time, strings don't change. And this would mean all this management is not necessary. However, the entire script is written assuming the strings can change. So, please use these helper functions.

```
script_gui_allocate_string(&defs[i].ComboBox.Hint, "");
script_gui_allocate_string(&defs[i].ComboBox.Units, "");
defs[i].ComboBox.Highlighted = 0;
defs[i].ComboBox.Enabled = 1;
i++;
```

This last increment is absolutely necessary to make sure the next block of initialization code works on the next control definition.

### 7.2.8 *Initializing a numeric edit*

A numeric edit is a GUI box that allows the user to type in a value.

```
defs[i].Type = WvNumericEdit;
```

Initialize the type.

```
defs[i].NumericEdit.Min = 0;
defs[i].NumericEdit.Max = 63;
```

Set the min/max values.

```
defs[i].NumericEdit.Value = 0;
```

Initialize the initial value

```
defs[i].NumericEdit.DecimalPlaces = 0;
```

Set the number of digits after the decimal point for display.

```
defs[i].NumericEdit.Type = WvInteger;
```

Set the type of value typed into the box. This can be an integer or a number in hex format. The declaration for this enum is in `wvdl_defs.h`.

```
script_gui_allocate_string(&defs[i].NumericEdit.Caption, "User Test Pattern Register 0");
script_gui_allocate_string(&defs[i].NumericEdit.Hint, "Enter an integer value for the
User Test Pattern Register 0");
script_gui_allocate_string(&defs[i].NumericEdit.Units, "");
```

Use the helper functions for the strings.

```
defs[i].NumericEdit.Highlighted = 0;
defs[i].NumericEdit.Enabled = 1;
i++;
```

Make sure the index is updated.

### 7.2.9 *adc14ds105\_tab0\_refresh*

This function looks at the current state of the hardware and refreshes the control definitions if necessary. The intent of this function is only to refresh the control definitions. This function should not manipulate the hardware. It should only look at the current state of the registers in the DUT-specific data structure. The hardware manipulation occurs when the GUI element is changed.

```

if (notify_change) {
    for (i = 0; i < num_defs; i++) {
        data->changed[i] = false;
    }
}

```

This little loop initializes the `changed` array. This array will keep track of which elements have changed. This array will then be consulted at the end of the function to determine if a change needs to be propagated back up to the DLL.

### 7.2.10 Refreshing a combo box

Since the combo box is a complicated, its refresh process is also a little more complicated than the other control definitions.

```

script_util_extract_bits(
    data->dev_ctrl,
    ADC14DS105_DEV_CTRL_OM_P,
    ADC14DS105_DEV_CTRL_OM_S,
    &register_field_value
);

```

This utility function is a helper to extract the register field from the entire register. This function simply right shifts the value by `_P` and then extracts `_S` number of bits and places the result in `register_field_value`.

```

if (defs[i].ComboBox.Values[defs[i].ComboBox.ItemSelected] != register_field_value) {

```

`defs[i].ComboBox.Values` is an array that contains all of the register values associated with each item in the list. This array was created with `script_gui_combobox_create_list`. And it was filled in with each call to `script_gui_combobox_add_item`.

`defs[i].ComboBox.ItemSelected` is the current item selected.

`defs[i].ComboBox.Values[defs[i].ComboBox.ItemSelected]` is therefore retrieving the register value associated with the currently selected item.

The non-equality check in the "if" clause determines if the currently selected item in the GUI is different than the value stored in the hardware.

```

for (j = 0; j < defs[i].ComboBox.ItemCount; j++) {
    if (defs[i].ComboBox.Values[j] == register_field_value) {
        defs[i].ComboBox.ItemSelected = j;
        data->changed[i] = true;
        break;
    }
}

```

This for loop is attempting a reverse lookup. The script needs to find the item which has a value that matches the current register field value.

The `changed` array element must be updated to indicate that this element has changed.

```

if (j == defs[i].ComboBox.ItemCount) {
    // the loop counter reached the end of the items. This means
    // the register field value is not found in the list of values.
    // This will leave the GUI inconsistent. Return error.
}

```

```

        return script_process_return(
            INTERFACE_RET_ERROR,
            "adc14ds105_tab0_refresh: register field value not found"
        );
    }

```

This is a sanity check to make sure the current value in the register can properly be supported by the GUI.

```

    }
    i++;

```

The index is updated to handle the next control definition.

### 7.2.11 Refreshing a combo box for cheaters

The code above for refreshing combo boxes is piece of code that will work for whatever values are stored in the `values` array. However, the combo boxes for this example are clever in that the index and the value are the same. So, the simplified code can take advantage of this fact and checked only the `ItemSelected` value.

```

script_util_extract_bits(
    data->dev_ctrl,
    ADC14DS105_DEV_CTRL_DLC_P,
    ADC14DS105_DEV_CTRL_DLC_S,
    &selected
);
if (defs[i].ComboBox.ItemSelected != selected) {
    defs[i].ComboBox.ItemSelected = selected;
    data->changed[i] = true;
}
i++;

```

### 7.2.12 Refreshing a numeric edit

Most of the refresh handling for the controls looks like this. The comparison is done between the `Value` field of the control definition and the register.

```

if (defs[i].NumericEdit.Value != data->usr_test_reg0) {
    defs[i].NumericEdit.Value = data->usr_test_reg0;
    data->changed[i] = true;
}
i++;

```

### 7.2.13 Sending a change notification

If a change is detected, then a notification must be sent to the DLL. The DLL will then pass the information back to the GUI so that the GUI can update the elements on the screen.

```

/* Cycle through changed indicators and send updates if necessary */
if (notify_change) {
    for (i = 0; i < num_defs; i++) {
        /*
            tab_index and control_index are explicitly checked here to
            filter out redundant updates. These indices are set
            when the GUI informs the DLL the user has changed
            a GUI element. There is no need to tell the GUI that the

```

```

        element the GUI changed has changed.

        FIXME: This may be incorrect.
    */
    if (data->changed[i] &&
        (
            (defs[i].Type == WvLevel1Register) ||
            ((tab_index != this_tab) || (control_index != i))
        )
    ) {
        script_request_gui_control_changed(this_tab, i, &defs[i]);
        script_request_gui_control_changed(this_tab, i, &defs[i]);
    }
}

```

This loop will cycle through the `changed` array looking for any of the values set to true. If it finds any, then it will send a request to the DLL via `script_request_gui_control_changed`.

The additional conditions in the `if` statement are trying to filter out redundant calls. The reason for the `FIXME` is that there can be situations where the hardware register field will react immediately upon the field changing. For instance, some bits are self-clearing by the hardware, meaning the hardware may automatically clear a bit once set.

If this occurs, the GUI will need to be updated so this self-clearing action is reflected. However, the additional checks will prevent this. This needs to be sorted out.

However, for level 1 elements, always pass the change back because a read action on a level 1 element will change the value.

### 7.2.14 *adc14ds105\_init\_tab1*

This contains the code to initialize tab 1. Most of the content is the same as the tab 0, except for the changes to the variables in the beginning and the actual initialization of the control definitions specific to tab 1.

### 7.2.15 *Initializing a Level 1 Register definition*

```
defs[i].Type = WvLevel1Register;
```

Set the type.

```
defs[i].Level1Register.Address = ADC14DS105_DEV_CTRL_ADDR;
```

use the constant from the register constants to set the address.

```
script_gui_allocate_string(&defs[i].Level1Register.Hint, "Device Control Register");
```

Use the string helpers to allocate the string.

```
defs[i].Level1Register.Value = data->dev_ctrl;
```

Set the initial value.

```
defs[i].Level1Register.DataWidth = WV_BIT_SIZE_8;
defs[i].Level1Register.AddressWidth = WV_BIT_SIZE_8;
```

Set the register bit sizes.



```
defs[i].Level1Register.RegisterType = WV_REGISTER_TYPE_READ_WRITE;
```

Set the type of register. The types are defined in `wvdl_defs.h`

```
defs[i].Level1Register.Action = WV_REGISTER_ACTION_READ;
```

This is simply initializing the `Action` field to a reasonable value. The `Action` field is actually an input to the script, used by the GUI to inform the script the type of action the user has requested.

```
i++;
```

Increment for the next control definition.

### 7.2.16 *adc14ds105\_tab1\_refresh*

This is refreshing all the elements in tab 1. This has the exact same format as tab 0, except for the tab 1 specific control definitions and constants.

### 7.2.17 *Refreshing a level 1 register*

```
if (defs[0].Level1Register.Value != data->dev_ctrl) {
    defs[0].Level1Register.Value = data->dev_ctrl;
    data->changed[i] = true;
}
i++;
```

This checks the value in the control against the hardware register and updates the `changed` array if necessary.

All the code to send the request is the same as tab 0.

### 7.3 *MANDATORY DUT FUNCTIONS - General, destroy*

This call cleans up any of the structures that were allocated. Right now, the only statements that should exist are the statements the clean up the control definitions.

```
script_gui_fini_defs(ADC14DS105_TAB0_NUM_DEFS, data->tab0_def);
script_gui_fini_defs(ADC14DS105_TAB1_NUM_DEFS, data->tab1_def);
```

There should be one call for each of the tabs.

### 7.4 *MANDATORY DUT FUNCTIONS - General, execute\_dut\_initialization*

This is finally where the very first hardware initialization can occur.

```
script_request_mm_wr(false, 0xd022, 0x10);
```

This writes a value to the Cypress memory space.

```
spi_auto_mode_wr(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, data->dev_ctrl);
spi_auto_mode_wr(0, 0, 8, ADC14DS105_USR_TEST_REG0_ADDR, 8, data->usr_test_reg0);
spi_auto_mode_wr(0, 0, 8, ADC14DS105_USR_TEST_REG1_ADDR, 8, data->usr_test_reg1);
```

These calls write values out to the ADC using the SPI functionality built into the FPGA. All WV5 compatible FPGAs must implement this SPI functionality. For this reason, there is no reason to bit-bang the SPI I/O lines and thus there is no need for custom code in the script files to explicitly execute each cycle of the SPI access.

This function is defined in `spi_auto_mode.h` and `spi_auto_mode.cpp`.

The only detail the auto mode cannot handle is the physical characteristics of the SPI lines. The auto mode code assumes these configurations have already been set. These characteristics include whether the data I/O lines are bi-directional or not and the speed of the interface.

This is the reason for the very first `script_request_mm_wr` call. This call is setting the SPI configuration register in the FPGA.

Each of these calls has a corresponding read command. So, instead of forcing default values into the FPGA, the script could read the values from the hardware to initialize its DUT-specific data structure. For instance,

```
spi_auto_mode_rd(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, &data->dev_ctrl);
```

This call would read the current state of the DEV\_CTRL register from the hardware and set the `data->dev_ctrl` to match.

## 7.5 *MANDATORY DUT FUNCTIONS - General, get\_clock\_frequency*

This function is called by the DLL when the DLL needs to acquire the most current sampling frequency detected by the FPGA.

```
script_request_get_fpga_clock_counts(&counts);
```

This helper function retrieves the raw FPGA counts. This is the value that is returned from the S\_RATE registers in the FPGA.

```
freq = counts * (8000.0*1/7);
```

This converts the raw counts to an actual frequency.

```
return script_process_return(
    script_set_return_clock_frequency(freq),
    "wv5_xc4vlx25_adc14ds105_get_clock_frequency"
);
```

script\_set\_return\_clock\_frequency returns the frequency to the DLL.

## 7.6 *MANDATORY DUT FUNCTIONS - General, poll\_hook*

This is generally not used.

This is useful for the scripts that require asynchronous checking or polling of the hardware.

## 7.7 *MANDATORY DUT FUNCTIONS - GUI, gui\_tab\_enum*

This returns the number of tabs and the tab structure array back to the FPGA. This data is eventually passed back to the GUI.

```
return script_process_return(
    script_set_return_gui_tab_enum(ADC14DS105_NUM_TABS, data->tabs),
    "wv5_xc4vlx25_adc14ds105_gui_tab_enum"
);
```

## 7.8 *MANDATORY DUT FUNCTIONS - GUI, gui\_control\_enum*

This returns the control definitions for each of the tabs.

```
script_get_params_gui_control_enum(&tab_index);
```

This call retrieves the parameters for the API function. In this case, the parameter is the tab index which will be used to determine which array of control definitions to return.

```
switch (tab_index) {
case 0:
    script_set_return_gui_control_enum(ADC14DS105_TAB0_NUM_DEFS, data->tab0_def);
```

This returns the number of control definitions and the array of control definitions for tab 0.

```
case 1:
    script_set_return_gui_control_enum(ADC14DS105_TAB1_NUM_DEFS, data->tab1_def);
```

This returns the number of control definitions and the array of control definitions for tab 1.

## 7.9 *Forward declarations for supporting GUI changes*

This section declares the functions that will be called when a GUI change needs to be handled. These are implemented later in the script.

```
Interface_Ret adc14ds105_tab0_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def);
Interface_Ret adc14ds105_tab1_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def);
```

## 7.10 *MANDATORY DUT FUNCTIONS - GUI, gui\_control\_change*

This is the general handler for processing the GUI changes. The main goal of this function is to call the proper tab-specific handler and then refresh the GUI to see if the changes had any other side effects on other GUI elements.

```
script_get_params_gui_control_change(&tab_index, &control_index, &control_def);
```

This retrieves the details for which GUI element changed.

```
switch (tab_index) {
case 0:
    ret = adc14ds105_tab0_change(data, control_index, control_def);
```

This calls the GUI change handler for tab 0.

```
case 1:
    ret = adc14ds105_tab1_change(data, control_index, control_def);
```

This calls the GUI change handler for tab 1.

```
ret = adc14ds105_tab0_refresh(data, true, tab_index, control_index);
```

Once the change has been processed, all of the GUI elements are refreshed, starting with tab 0.

The reason all of the GUI elements are refreshed is because the GUI change may affect other GUI elements.

For instance, assume one of the GUI elements is a button which is supposed to reset all registers to the default state. Once this GUI element is changed, all the hardware registers can potentially change.

And if all the hardware registers change, then the GUI elements which depend on those registers will need to be updated.

By refreshing all of the GUI elements, all of these potentially chained consequences are handled automatically.

```
ret = adc14ds105_tab1_refresh(data, true, tab_index, control_index);
```

This refreshes the GUI elements for tab 1.

### 7.10.1 *adc14ds105\_tab0\_change*

This handles the change in a GUI element within tab 0.

```
script_gui_copy_def(&defs[control_index], new_def);
```

All GUI change handlers should have this at the beginning of the function. This helper function handles all the details regarding copying the data from the new control definition into the definition in the DUT-specific structure.

The reason for the helper function is to hide the details of memory copying involving in copying arrays. When this new definition is passed into the script, the script cannot assume the new definition will be persistent. The new definition is owned by the GUI, and not the DLL.

This means all the strings and the arrays associated with the combo boxes must be "deep copied". Since the string sizes and array sizes may change, the strings/arrays must be deallocated, then reallocated with the new size, and then the contents from the new definition copied into these newly allocated locations.

### 7.10.2 *Changes on elements other than level 1 registers*

```
switch (control_index) {
case 0:
```

This case statement handles the definition at index 0.

```
    script_util_set_bits(
        new_def->ComboBox.Values[new_def->ComboBox.ItemSelected],
        ADC14DS105_DEV_CTRL_OM_P,
        ADC14DS105_DEV_CTRL_OM_S,
        &data->dev_ctrl
    );
```

This helper function takes the new value and sets the bits in the register defined by the register field.

The GUI will have modified `new_def->ComboBox.ItemSelected` to indicate that the user has selected a new item in the combo box. `new_def->ComboBox.Values` is an array that contains all the values associated with each item in the list. The `_P` and `_S` are used to define the bits in the register that make up this field. In this case, the field is the "OM" field or the "Operational Mode" field. And finally, the last parameter is the register in the DUT-specific data structure to modify.

```
    spi_auto_mode_wr(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, data->dev_ctrl);
```

The `script_util_set_bits` only modified the copy of the register in the DUT-specific data structure. The actual value still needs to be written out. `spi_auto_mode_wr` is the call that will finally manipulate the hardware.

These two calls are repeated several times for each of the GUI elements. This is the most common type of GUI change handling.

The reason is most of the time, one GUI element corresponds to exactly one field in a hardware register; when a GUI element changes, only one field of a hardware register changes. So, on a change, the DUT-specific copy of the register is updated and then that copy is written out to the hardware.

### 7.10.3 *adc14ds105\_tab1\_change*

This handles the change for tab 1.

### 7.10.4 *Changes on level 1 elements*

```
if (new_def->Level1Register.Action == WV_REGISTER_ACTION_READ) {
    WvWord temp_read;

    // Clear out register.
    data->dev_ctrl = 0;

    spi_auto_mode_rd(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, &temp_read);

    // Use the bit helper to filter out the top bits (just in case they were non-zero).
    script_util_set_bits(
        temp_read,
        0,
        8,
        &data->dev_ctrl
    );
} else {
    data->dev_ctrl = new_def->Level1Register.Value;
    spi_auto_mode_wr(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, data->dev_ctrl);
}
```

The difference with handling changes for level 1 elements is that level 1 elements contain a command for either reading or writing the associated hardware register.

The read command will force the script to read the hardware register. And the write command will force the script to write out the hardware register.

### 7.11 *MANDATORY DUT FUNCTIONS - Capture, get\_capture\_parameters*

This returns the capture configuration parameters.

```
return script_process_return(
    script_set_return_capture_parameters(ADC14DS105_NUM_CAP_CONFIGS, data->cap_config,
    data->priv_cap_config),
    "wv5_xc4vlx25_adc14ds105_get_capture_parameters"
);
```

These parameters have already been initialized so this function simply passes the values back to the DLL using the shared API.

### 7.12 *MANDATORY DUT FUNCTIONS - Capture, execute\_capture\_actions*

This function can be used to do custom commands during the capture sequence. Most of the time, there is no need for any custom commands because the DLL handles all the details.

### 7.13 *MANDATORY DUT FUNCTIONS - Capture, execute\_post\_capture\_actions*

This function can be used to post-process the data. Most of the time, there is no need for this since the DLL already handles the de-interleaving of the data for WV5 compatible FPGA images.

### 7.14 *MANDATORY DUT FUNCTIONS - DAC data download parameters*

This is DAC specific and not used for ADCs.

### 7.15 *MANDATORY DUT FUNCTIONS - DAC download command process*

This is DAC specific and not used for ADCs.

### 7.16 *MANDATORY DUT FUNCTIONS - General access, general\_access\_rd*

This function is called when the user attempts to access the hardware using the hardware access panel in the GUI.

For WV5, most of the accesses require the same code and so all of the code is in the board-level script.

### 7.17 *MANDATORY DUT FUNCTIONS - General access, general\_access\_wr*

This function is called when the user attempts to access the hardware using the hardware access panel in the GUI.

For WV5, most of the accesses require the same code and so all of the code is in the board-level script.

Rule of thumb for strings in the GUI:

For tab structures, just use a constant

For control definitions, always use `script_gui_allocate_string`

Exercises – what exercises are necessary? GUI changes examples.

Creating the file

Assuming a board script already exists, the only new file necessary is the the .cpp file. The recommended method of creating this file is to simply copy and rename an existing .cpp file, choosing one whose hardware is similar to the new hardware.

Naming the file

Please see the naming convention document (naming.doc) and the image\_map.xml document (image\_map.doc). In both of these documents, the area to focus on is how the DUT script names are chosen. Although these documents assume the XML scripts (with extension .xml), the same rules apply for C scripts (with the .cpp extension).

In general, the script names are created with the pattern "board\_fpga\_dut.dut.cpp", where "board" is the board name, "fpga" is the FPGA model, and "DUT" is the name of the DUT. Most of the time, the "DUT" part can be renamed to match the new hardware.

Prep work

The first step after renaming the file is to do a very careful "search and replace".



## 8 Typical Customization Issues

This section explains how to implement several specific customizations that a given hardware/software package would typically require.

### 8.1 Customizing the DUT hardware initialization

After the board and DUT are initialized, the hardware may need some initialization. Sometimes some of the front-end specific register needs to be modified. Other times the DUT registers may need some initialization.

This hardware initialization should be implemented in the mandatory function `execute_dut_initialization`.

In the example script, there are four steps in the initialization:

```
script_request_mm_wr(false, 0xd022, 0x10);
```

This initializes the SPI\_CFG register.

```
spi_auto_mode_wr(0, 0, 8, ADC14DS105_DEV_CTRL_ADDR, 8, data->dev_ctrl);
spi_auto_mode_wr(0, 0, 8, ADC14DS105_USR_TEST_REG0_ADDR, 8, data->usr_test_reg0);
spi_auto_mode_wr(0, 0, 8, ADC14DS105_USR_TEST_REG1_ADDR, 8, data->usr_test_reg1);
```

These three write registers on the ADC.

In general, these two functions are all that are required to initialize the hardware.

`script_request_mm_wr` should be used to write all FPGA registers that need some configuration. Most of the time, this is used to configure the front-end specific registers of the FPGA.

- First parameter should always be false.
- Second parameter should be the FPGA register address as seen from the Cypress
- Third parameter is value to write

`spi_auto_mode_wr` should be used to write ADC registers.

- First parameter should be 0
- Second parameter should be 0
- Third parameter is the number of bits in the address
- Fourth parameter is the register address
- Fifth parameter is the number of bits in the data
- Sixth parameter is the data

### 8.2 Modifying frequency calculation

This requires a change to `wv5_xc4vlx25_adc14ds105_get_clock_frequency`.

```
freq = counts * (8000.0*1/7);
```

This line needs to be modified to apply custom calculations to translate from raw FPGA counts to a sampling frequency.

## 8.3 Adding a tab

This will involve several steps, but the general idea is to copy everything that is labeled with "tab 0" and replace the "0" with the new tab index.

The current example has two tabs, tab0 and tab1. Assume a third tab, tab 2, needs to be created.

```
#define ADC14DS105_TAB0_NUM_DEFS 9
#define ADC14DS105_TAB1_NUM_DEFS 1
#define ADC14DS105_TAB2_NUM_DEFS xxxxx
```

Add a line for the number of definitions in the tab.

```
#define ADC14DS105_MAX_DEFS ADC14DS105_TAB0_NUM_DEFS
```

This constant may need to be modified if the new tab has more than 9 control definitions.

```
WvControlDefinition tab0_def[ADC14DS105_TAB0_NUM_DEFS];
WvControlDefinition tab1_def[ADC14DS105_TAB1_NUM_DEFS];
WvControlDefinition tab2_def[ADC14DS105_TAB2_NUM_DEFS];
```

Add a line for the new array of tab definitions.

```
Interface_Ret adc14ds105_init_tab0(Adc14ds105_Data* data);
Interface_Ret adc14ds105_init_tab1(Adc14ds105_Data* data);
Interface_Ret adc14ds105_init_tab2(Adc14ds105_Data* data);
```

```
Interface_Ret adc14ds105_tab0_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
);
```

```
Interface_Ret adc14ds105_tab1_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
);
```

```
Interface_Ret adc14ds105_tab2_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
);
```

Add a new line for initializing the tab and refreshing the tab.

```
ret = adc14ds105_init_tab0(data);
if (ret != INTERFACE_RET_OK) {
    return ret;
}

ret = adc14ds105_init_tab1(data);
if (ret != INTERFACE_RET_OK) {
    return ret;
}

ret = adc14ds105_init_tab2(data);
if (ret != INTERFACE_RET_OK) {
    return ret;
}
```

Add a call to initialize the tab.

```
Interface_Ret adc14ds105_init_tab2(Adc14ds105_Data* data) {
```

```

WvWord num_defs = ADC14DS105_TAB2_NUM_DEFS;
WvControlDefinition* defs = data->tab2_def;
WvWord i;

script_debug_out_pre("adc14ds105_init_tab2");
script_debug_out_endl();

data->tabs[2].caption = "XXX YYY ZZZ";
data->tabs[2].level = 2;
data->tabs[2].tags = "";

script_gui_init_defs(num_defs, defs);

i = 0;

/*
   Add new definitions
*/

if (i != num_defs) {
    return script_process_return(
        INTERFACE_RET_ERROR,
        "adc14ds105_init_tab2: control definition counts do not match"
    );
}

/* Refresh all the values in the GUI */
return script_process_return(
    adc14ds105_tab2_refresh(data, false, 0, 0),
    "adc14ds105_init_tab2"
);
}

```

Add the code to initialize the tab. The only changes from the tab 0 code have been to change the `num_defs` and `defs` variables to make sure they use tab 2 data, the `tabs[2]` index, and the strings for the debug output.

The major change is the removal of the tab 0 specific control definitions. These will have to be added.

```

Interface_Ret adc14ds105_tab2_refresh(
    Adc14ds105_Data* data,
    bool notify_change,
    WvWord tab_index,
    WvWord control_index
) {
    WvWord i;
    WvWord j;
    WvWord num_defs = ADC14DS105_TAB2_NUM_DEFS;
    WvControlDefinition* defs = data->tab2_def;
    WvWord this_tab = 2;

    script_debug_out_pre("adc14ds105_tab2_refresh: notify:%d", notify_change);
    script_debug_out_endl();

    if (notify_change) {
        for (i = 0; i < num_defs; i++) {
            data->changed[i] = false;
        }
    }

    i = 0;

    /*
       Add refresh code
    */

    if (i != num_defs) {
        return script_process_return(

```

```

        INTERFACE_RET_ERROR,
        "adc14ds105_tab2_refresh: control definition counts do not match"
    );
}

/* Cycle through changed indicators and send updates if necessary */
if (notify_change) {
    for (i = 0; i < num_defs; i++) {
        if (data->changed[i] &&
            (
                (defs[i].Type == WvLevel1Register) ||
                ((tab_index != this_tab) || (control_index != i))
            )
        ) {
            script_request_gui_control_changed(this_tab, i, &defs[i]);
        }
    }
}

return script_process_return(
    INTERFACE_RET_OK,
    "adc14ds105_tab2_refresh"
);
}

```

Add code to refresh the tab. Again the only change has been to `num_defs`, `defs`, and `this_tab` in the beginning of the function.

However, the control-specific code must be added.

```

script_gui_fini_defs(ADC14DS105_TAB0_NUM_DEFS, data->tab0_def);
script_gui_fini_defs(ADC14DS105_TAB1_NUM_DEFS, data->tab1_def);
script_gui_fini_defs(ADC14DS105_TAB2_NUM_DEFS, data->tab2_def);

```

Add code to clear out the definitions.

```

case 0:
    script_set_return_gui_control_enum(ADC14DS105_TAB0_NUM_DEFS, data->tab0_def);
    break;
case 1:
    script_set_return_gui_control_enum(ADC14DS105_TAB1_NUM_DEFS, data->tab1_def);
    break;
case 2:
    script_set_return_gui_control_enum(ADC14DS105_TAB2_NUM_DEFS, data->tab2_def);
    break;

```

Add code to handle the control enumeration request.

```

Interface_Ret adc14ds105_tab0_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def);
Interface_Ret adc14ds105_tab1_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def);
Interface_Ret adc14ds105_tab2_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def);

```

Add a declaration for the tab 2 change handler.

```

case 0:
    ret = adc14ds105_tab0_change(data, control_index, control_def);
    break;
case 1:
    ret = adc14ds105_tab1_change(data, control_index, control_def);
    break;
case 2:
    ret = adc14ds105_tab2_change(data, control_index, control_def);
    break;

```

Add code to call the handler.

```
ret = adc14ds105_tab0_refresh(data, true, tab_index, control_index);
if (ret != INTERFACE_RET_OK) {
    return script_process_return(
        ret,
        "adc14ds105_tab0_refresh"
    );
}
ret = adc14ds105_tab1_refresh(data, true, tab_index, control_index);
if (ret != INTERFACE_RET_OK) {
    return script_process_return(
        ret,
        "adc14ds105_tab1_refresh"
    );
}
ret = adc14ds105_tab2_refresh(data, true, tab_index, control_index);
if (ret != INTERFACE_RET_OK) {
    return script_process_return(
        ret,
        "adc14ds105_tab2_refresh"
    );
}
```

Add code to refresh the tab after any changes.

```
Interface_Ret adc14ds105_tab2_change(Adc14ds105_Data* data, WvWord control_index, const
WvControlDefinition* new_def) {
    WvControlDefinition* defs = data->tab2_def;

    script_debug_out_pre("adc14ds105_tab2_change");
    script_debug_out_endl();

    script_gui_copy_def(&defs[control_index], new_def);

    switch (control_index) {
        /*
         * Add per-control handlers.
         */
        default:
            return script_process_return(
                INTERFACE_RET_ERROR,
                "adc14ds105_tab2_change: index out of range"
            );
    }

    return script_process_return(
        INTERFACE_RET_OK,
        "adc14ds105_tab2_change"
    );
}
```

Add implementation for the change handler. Again, the variables at the beginning of the function are changed to match tab 2 and the debug strings have changed.

The major part missing part is the control-specific actions.

## 8.4 Adding a control to an existing tab

Since the functions are already defined, the changes required involve adding code to the existing functions.

But first, the constant representing the number of control definitions must change.

Assume the example script and assume one more numeric edit for tab 0 is being added.

```
#define ADC14DS105_TAB0_NUM_DEFS 10
```

## Increment number of definitions

```
#define ADC14DS105_MAX_DEFS ADC14DS105_TAB0_NUM_DEFS
```

This constant may need to change.

```
/* Definition 9 */
defs[i].Type = WvNumericEdit;
defs[i].NumericEdit.Min = 0;
defs[i].NumericEdit.Max = 255;
defs[i].NumericEdit.Value = 0;
defs[i].NumericEdit.DecimalPlaces = 0;
defs[i].NumericEdit.Type = WvInteger;
script_gui_allocate_string(&defs[i].NumericEdit.Caption, "additional reg");
script_gui_allocate_string(&defs[i].NumericEdit.Hint, "additional hint");
script_gui_allocate_string(&defs[i].NumericEdit.Units, "additional units");
defs[i].NumericEdit.Highlighted = 0;
defs[i].NumericEdit.Enabled = 1;
i++;
```

Edit `adc14ds105_init_tab0` to include this new numeric edit.

```
/* Definition 9 */
if (defs[i].NumericEdit.Value != data->some_new_register) {
    defs[i].NumericEdit.Value = data-> some_new_register;
    data->changed[i] = true;
}
i++;
```

Edit `adc14ds105_tab0_refresh` to refresh the element from a register.

```
case 9:
    script_util_set_bits(
        (WvWord)new_def->NumericEdit.Value,
        ADC14DS105_SOME_NEW_REGISTER_SOME_NEW_FIELD_P,
        ADC14DS105_SOME_NEW_REGISTER_SOME_NEW_FIELD_S,
        &data->some_new_register
    );
    spi_auto_mode_wr(0, 0, 8, ADC14DS105_SOME_NEW_REGISTER_ADDR, 8, data-
>some_new_register);
    break;
```

Edit `adc14ds105_tab0_change` to handle a change to the element.

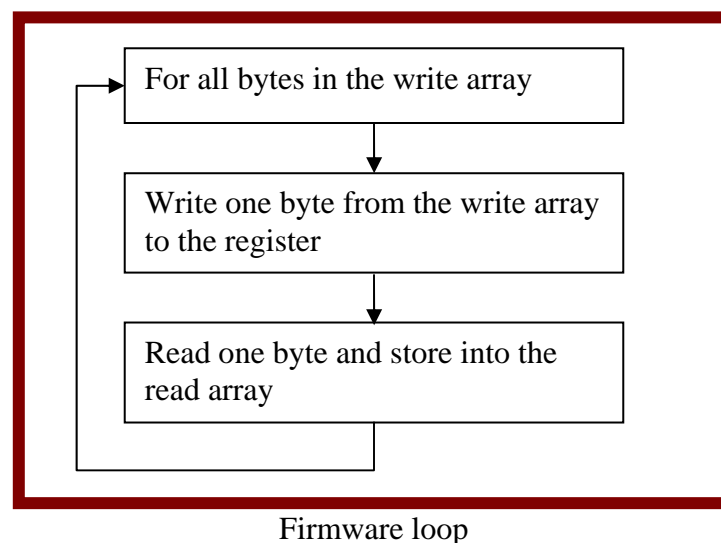
## 9 Soft Serial Protocol

The WV5 Core software provides a means to directly control the controller card's USB MCU's ports from the script file. This way a standard controller card can be used to interface with a variety of devices by adapting only the script file - which is a basic feature of the WV5 system. Precision System Product Group's SensorPath platform employs this technique. The SensorPath platform uses the USI-2 controller card to carry out this function. This section describes how one may program the script file to adapt the Soft Serial protocol technique to match the serial interface protocol of the device of interest.

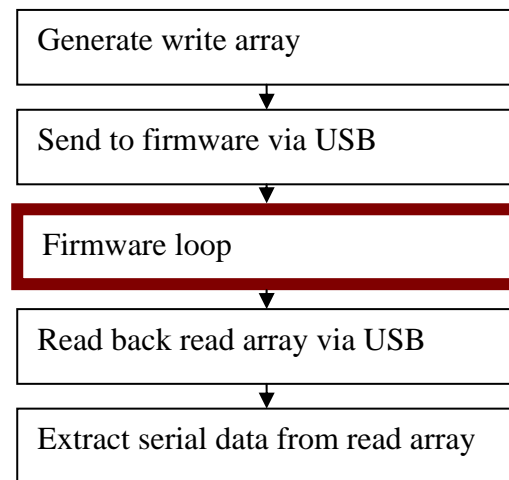
### 9.1 Overview

The main idea is to keep the firmware that generates the serial protocol as simple as possible. This has two effects. First, the simple firmware should allow for maximum throughput since the firmware will not waste any cycles doing unnecessary work. Second, the protocol design will be transferred to the host so that host will have as much flexibility as possible.

To do this, the firmware is reduced to writing values to a specific register from an array of bytes. And the firmware will read back the register after every single write.



The host's responsibility is to architect an array of bytes that will result in the intended serial protocol. The host will send this byte array to the firmware via USB, the firmware will execute the above loop, the firmware will return the read array back to the host via USB, and finally the host will interpret the read array to extract the serial data.



## 9.2 Serial Protocol Array (write array)

Please read the general overview regarding SensorPath first. If the hardware specification has been followed, then there should be two serial communication buses on Port B of and two serial communication buses on Port D of the microcontroller (Cypress EZ-USB on the USI-2 card).

The fact that all the lines for one serial communication bus are wholly contained on a single Cypress GPIO port is important.

The serial protocol array is an array of bytes that are to be written to the Cypress GPIO port. There should be one byte for every single edge of the clock. Two bytes are required to generate one single clock pulse.

These byte values are written directly to the Cypress GPIO port with no filtering applied by the firmware. This means each byte has the potential to manipulate two serial communication buses. This must be taken into consideration when designing the serial protocol array.

The best method to deal with this situation is to always deassert bus select line for the bus that is not being used accessed.

## 9.3 Firmware Return Array

Since all the lines for a serial communication bus are wholly contained on one Cypress GPIO port, the output data pin of the device should be connected to a GPIO pin on the same port as all the other signals.

After the firmware writes one value from the serial protocol array, the firmware will read the port back and place the result in the firmware return array. The firmware does not do any



processing of the port value; it simply copies the value into the firmware return array. This means the firmware return array will contain data from two serial buses. It is the host's responsibility when processing the firmware return array to disregard the port that is not being accessed.

If the device is generating data, then reading the port will capture the current value on the device output/Cypress input pin.

If the device is not generating data, then the host can ignore this value of the firmware return array.

Note that the firmware does not selectively choose when to read the Cypress port. The firmware always reads back the Cypress port. This means the firmware return array should be exactly the same size as the serial protocol array.

The host will process the entire firmware return array and it is the host's responsibility to disregard the read values that do not contain any device data.

## 9.4 Examples

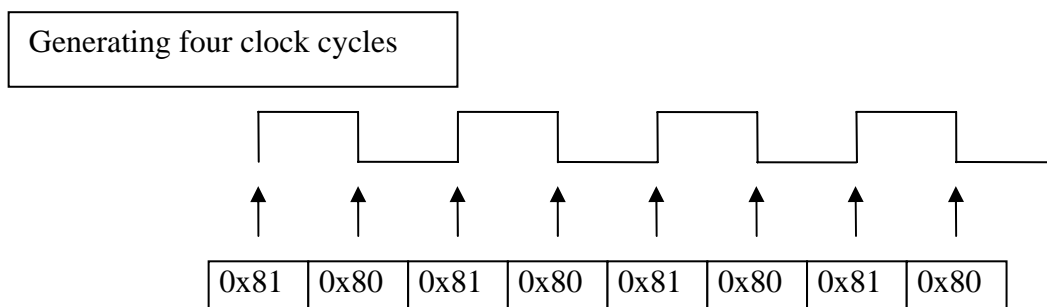
### 9.4.1 Example 1

In this example, attempt to create a serial protocol array to generate four clock cycles:

Active bus is bus A

Bus chip select is active low

Each clock cycle is first a rising edge and then a falling edge



Since the active bus is bus A, the lower four bits of Port B will be used. Since bus B is not used, the bus select is deasserted. Each value contains a different edge of the clock.

So, the serial protocol array is:

Array[0] = 0x81

Array[1] = 0x80

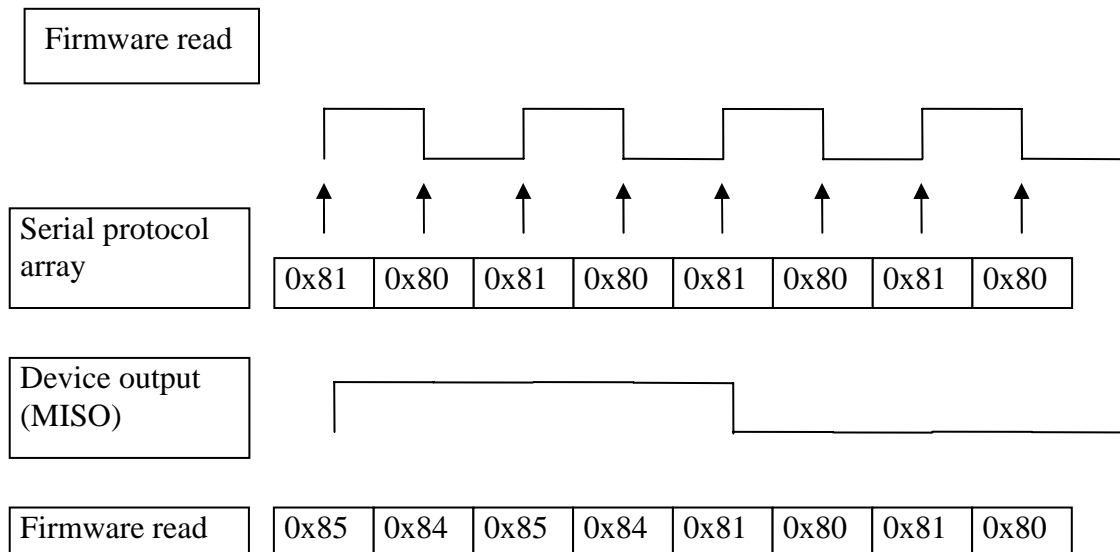
Array[2] = 0x81

Array[3] = 0x80

Array[4] = 0x81

```
Array[5] = 0x80
Array[6] = 0x81
Array[7] = 0x80
```

### 9.4.1 Example 2



This example is an extension of example 1. Again, assume:

- Active bus is bus A
- Bus chip select is active low
- Each clock is a rising edge then a falling edge

Since the active bus is bus A, port B on the Cypress will be used.

This time, assume the device will generate data. Assume:

- New bit value is generated at the rising edge of the clock
- New data is generated with MSB out first
- New data is binary 1100

The firmware will read port B after writing out one element of the serial protocol array. Bit 2 of Port B will change according to the new bit generated by the device. Since the output pins of the port remain the same, the read will effectively be the most current state of the output pins logically "OR"ed with the device output (master input) at bit 2.

The elements of the firmware return array are populated as the firmware reads the port. So, when the firmware has completed the writing out the serial protocol array, it should return the following firmware return array back to the host:

```
Firmware return array[0] = 0x85
Firmware return array[1] = 0x84
Firmware return array[2] = 0x85
```

```
Firmware return array[3] = 0x84
Firmware return array[4] = 0x81
Firmware return array[5] = 0x80
Firmware return array[6] = 0x81
Firmware return array[7] = 0x80
```

## 9.5 *Software – General flow of the code*

The following is a very high-level description of the process of communicating with a device using a custom serial protocol. "client" in this case is the DLL client, most typically the GUI.

Hardware initialization:

- Board powered up
- USB cable plugged in

Software initialization:

- Client initializes DLL
- DLL detects a board plugged in
  - DLL consults VID/PID and determines the board
  - DLL loads board-level script
  - DLL initializes board
- DLL reports the SensorPath2008 board
- DLL reports 1 unnamed DUT
- Client retrieves list of DUTs supported on this board
- Client decides which DUT is to be used
- Client informs DLL of which DUT to be used
- DLL looks for DUT script
- DLL initializes DUT

Communication:

- Client attempts to access memory address that maps to SensorPath reserved address
- DLL processes request
- DLL calls script file to process memory access
- Script processes memory access
- Script determines that the memory access is at SensorPath reserved address
- Script determines which configuration to use by subtracting SensorPath reserved base address from the requested address
- Script customizes serial protocol array for channel selection or register address (if necessary)
- Script ends serial protocol array to firmware
- Firmware executes array
- Firmware returns the firmware return array
- Script processes firmware return array
- Script extracts data bit-by-bit from the firmware return array
- Script returns data to DLL

DLL returns data to client

## 9.6 Software Specifics - DLL

Most of the logic to support this type of custom serial communication is in the scripts. This means the communication can be altered without having to recompile any of the software.

Please review all of the C scripting documentation, the WV5 System Designers' Guide and the SensorPath hardware documentation.

### 9.6.1 *image\_map.xml*

This file contains naming exceptions to map a particular DUT to a particular support file. A support file is either a CPU image (\*.bix), a FPGA image (\*.bit), or a script file (\*.xml or \*.cpp).

A DUT is usually identified by the DUT EEPROM that is present on the DUT board. The SensorPath boards are an exception; they have been explicitly designed not to have DUT EEPROMs.

For these boards, image\_map.xml serves an additional purpose. Image\_map.xml also contains a list of DUTs for boards that will not support DUT EEPROMs.

```
<board
  vid="0x0400"
  pid="0x3111"
  nick="sensorpath2008"
  friendly_name="SensorPath"
  cpu="cy7c68013a"
  fpga="none"

  duts="DIFFERENTIAL_ALLINONE,OPTICAL_ENC,SP1202S01RB,SP1202S02RB,SP1202S03RB,SP1202S05RB,SP1602S01RB"
  dut_descs="Differential Sensor Board,Optical Encoder,Differential Sensor Board
[ADC121S101],Dual Channel Thermocouple Board [ADC122S101],Photo Diode Board
[ADC122S101],Thermocouple Sensor Board Version 2 [ADC122S101],High Resolution
Differential Sensor Board [ADC161S626]"
/>
```

This is the current listing for SensorPath boards. The `dut` and `dut_descs` sections are specific to SensorPath boards. The `dut` section is a comma separated list of all the code names for all the DUTs supported. The `dut_descs` section is a comma separated list of more meaningful descriptions for the DUTs supported. Both of these sections must be aligned with each other, meaning the ordering must match between the two.

The code name in the `dut` section will be used as the DUT name. The DUT name will be used to find the exact script file necessary to support this device. Please consult naming.doc and sections 3 and 4 of "cscript – gen.doc".

Example:

DUT code name: DIFFERENTIAL\_ALLINONE

**Description:** Differential Sensor Board

**DUT name:** DIFFERENTIAL\_ALLNONE (this is exactly the same as DUT code name)

**Script file:** sensorpath2008\_none\_differential\_allnone.dut.cpp

### 9.6.2 Choosing a DUT

Since the DLL cannot predict which DUT the user has connected, the DLL must be explicitly told which DUT to use.

There are two functions in the WV DLL API that assist in this process.

```
WvBool Wv_API WvGetSupportedDuts(  
    WvWord    board_index,  
    WvString* return_dut_names,  
    WvString* return_dut_descriptions  
);
```

This will return the list of DUT codenames and DUT descriptions from the image\_map.xml file.

```
WvBool Wv_API WvSetDut(WvWord board_index, WvWord dut_index, WvString dut_name);
```

This will force the DLL to use name given in `dut_name` as the DUT name. `dut_name` must be an entry in the DUT code name list.

## 9.7 Software – Script

There are already several examples of how to create the serial protocol array and how to manipulate the firmware return array in the scripting environment. The current scripts that implement this are:

- sensorpath2008.brd.cpp
- sensorpath2008\_none\_optical\_encoder.dut.cpp
- sensorpath2008\_none\_sp1202s01rb.dut.cpp
- sensorpath2008\_none\_sp1602s01rb.dut.cpp

### 9.7.1 *sensorpath2008.brd.cpp*

This is a board-level script file and will get loaded every time a SensorPath board is plugged in. Since many of the DUTs share the same serial protocol array and process the firmware return array the same, the code was moved into the board-level script file so all other script files can share it.

Currently, the following script files use the board-level code:

- sensorpath2008\_none\_differential\_allinone.dut.cpp
- sensorpath2008\_none\_sp1202s02rb.dut.cpp
- sensorpath2008\_none\_sp1202s03rb.dut.cpp
- sensorpath2008\_none\_sp1202s05rb.dut.cpp

### 9.7.2 *Details of custom serial communication in sensorpath2008.brd.cpp*

The following is the serial protocol array:

```
const unsigned char sensorpath2008_brd_fast_spi_cmd[SENSORPATH2008_BRD_CMD_SZ] = {
    /*
     * Fast SPI command sequence header
     */
    CMD_ID_FAST_SPI,
    0,
    0,

    /*
     * Start raw port values
     */
    // Index 0: Make sure the chip select is not active
    0x89,
    0x89,
    0x89,
    // Index 3: Assert chip select before starting the clocks
    0x81,

    /*
     * Index 4: 16 clocks. This block will be manipulated when the
     * address is set. The data read in this phase will be ignored.
     */
    0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81,
    0x80, 0x81,
    0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81,
    0x80, 0x81,

    // Index 36: De-assert chip select and wait a bit before asserting chip select
    0x89,
    0x89,
    0x89,
    0x81,

    /*
     * Index 40: Read the data. The address field in this case is a "don't-care." The
     * data of interest is the actual conversion from the previously selected address.
     */
    0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81,
    0x80, 0x81,
    0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81, 0x80, 0x81,
    0x80, 0x81,
```

```
// Index 72: Done!
0x89
};
```

There is one detail regarding this array that differs slightly from Section 4 (Serial Protocol Array) and the examples. The actual data sent to the firmware from the host PC is command that has two parts, a command header and the actual array:

Command header – 3 bytes  
Serial Protocol Array – variable number of bytes

The serial protocol array must start right after the header.

The `index` comments within the array is an index into the serial protocol array section of the command. This way it is a little easier to see exactly how many bytes are used in the protocol and helps keep track of the index when dealing with large blocks of values that look exactly the same.

There are two constants in the header file that are associated with this command:

```
#define SENSORPATH2008_BRD_RAW_SZ 73
#define SENSORPATH2008_BRD_CMD_SZ 76
```

The `_RAW_SZ` value is the size of the serial protocol array. The `_CMD_SZ` value is the size of the entire command (header + serial protocol array = 3 + 73 = 76).

This serial protocol array deals only with the the lower 4 bits of the port. This means this can be used for both serial communication bus A and bus C. However, for this board, only bus A is used. Since bus B is shared on Port B, the chip select for bus B is kept deasserted.

### 9.7.3 *Details of the actual protocol*

This serial protocol array generates two bursts of 16 clocks. The first burst is used to program the channel to read and then the second burst is used to read the value.

Since these ADCs require 16 clocks for the conversion, the data returned in the first burst is ignored.

Each burst is preceded with a slight pause of four elements by de-asserting the chip select and then reasserting the chip select.

### 9.7.4 *Details of routine that sends the command and processes the firmware return array*

The routine that manipulates the device address for the serial protocol array and then processes the firmware return array is `sensorpath2008_brd_fast_spi_rd`. This routine is called by the other DUT scripts that all share the same serial protocol.

```
if (sscanf(config_str, "SPI%d_dev%d_ch%d", &spi_index, &dev_index, &ch) != 3) {
    return script_process_return(
```

```

        INTERFACE_RET_ERROR,
        "Configuration name [%s] is not properly formatted for SensorPath",
        config_str
    );
}

```

Each channel on each device on each bus has a unique identifier. This `sscanf` call is to parse out the physical parameters from the identifier.

```

    if (dev_index != 0) {
        return script_process_return(
            INTERFACE_RET_ERROR,
            "Configuration name [%s] is not properly formatted for SensorPath: Device ID
is out of range",
            config_str
        );
    }

    if (ch > 2) {
        return script_process_return(
            INTERFACE_RET_ERROR,
            "Configuration name [%s] is not properly formatted for SensorPath: Channel ID
is out of range",
            config_str
        );
    }
}

```

These are sanity checks to make sure the channel access can be handled by the serial protocol array. Currently, only one device is supported at index 0. And at most, two channels are supported.

```
memcpy(data->tx_buf, sensorpath2008_brd_fast_spi_cmd, SENSORPATH2008_BRD_CMD_SZ);
```

This initializes the buffer which is used to send the data. The added benefit to copying the entire command to the buffer is to allow modifications to the serial protocol array before sending it to the firmware. The serial protocol array needs to be modified because the channel selection must be programmed.

```

data_bit = sensorpath2008_get_data_bit_pos(spi_index, true);
mask = (1 << data_bit);

```

`sensorpath2008_get_data_bit_pos` returns the bit position of either the MOSI or MISO bit. In this case, the second parameter is `true` and so this will return the MOSI bit. If the hardware follows the spec, then for bus A and C, this should be bit 1 and for bus B and D this should be bit 5.

```

for (i = 0; i < SENSORPATH2008_BRD_ADDR_SZ; i++) {
    // Assume MSB first always
    new_bit = ch & (1 << (SENSORPATH2008_BRD_ADDR_SZ - i - 1));

    // Write twice for each cycle of the clock
    if (new_bit) {
        data->tx_buf[offset] |= mask;
        data->tx_buf[offset+1] |= mask;
    } else {
        data->tx_buf[offset] &= (~mask);
        data->tx_buf[offset+1] &= (~mask);
    }

    // Write the address to the second set of clock pulses
    data->tx_buf[36+offset] = data->tx_buf[offset];
    data->tx_buf[36+offset+1] = data->tx_buf[offset+1];
    offset += 2;
}

```

This loop places the value of `ch` into the serial output array. `offset` is the offset into the command and not a direct offset into the serial output array. The new bit is set into two consecutive bytes



in the serial output array because the bit will be held for the entire clock pulse. Also, the values at `offset+36` and `offset+36+1` are set because the second block of clock pulses need to share the same `ch` value.

```
interface_ret = script_request_bulk_usb_tx(1, SENSORPATH2008_BRD_CMD_SZ, data-
>tx_buf);
if (interface_ret != INTERFACE_RET_OK) {
    return script_process_return(
        interface_ret,
        "Fast SPI failed command execution"
    );
}
```

This sends the command to the firmware.

```
interface_ret = script_request_bulk_usb_rx(0, SENSORPATH2008_BRD_RAW_SZ, data-
>rx_buf);
if (interface_ret != INTERFACE_RET_OK) {
    return script_process_return(
        interface_ret,
        "Fast SPI failed reading back"
    );
}
```

This reads back the firmware read array.

```
data_bit = sensorpath2008_get_data_bit_pos(spi_index, false);
mask = (1 << data_bit);
```

This determines the bit position for the MISO bit. The second parameter is `false`.

```
offset = 40;
*ret = 0;    for (i = 0; i < SENSORPATH2008_BRD_DATA_SZ; i++) {
    new_bit = ((data->rx_buf[offset] & mask) ? 1 : 0);

    // Assume MSB first always
    (*ret) |= (new_bit << (SENSORPATH2008_BRD_DATA_SZ - i - 1));

    // Skip two because there are two port reads for each clock cycle.
    offset += 2;
}
```

This loop constructs the return data by extracting one bit from each entry in the firmware read array. The offset starts at 40 because the data is only valid after 40 elements in the serial protocol array. The 40 elements consist of 4 elements of pause followed by 32 elements for the clock transitions (16 clocks) followed by another 4 elements of pause.

### 9.7.5 Use of *sensorpath2008.brd.cpp* by other script files

As noted earlier, the following scripts utilize `sensorpath2008.brd.cpp` for the communication:

- `sensorpath2008_none_differential_allinone.dut.cpp`
- `sensorpath2008_none_sp1202s02rb.dut.cpp`
- `sensorpath2008_none_sp1202s03rb.dut.cpp`
- `sensorpath2008_none_sp1202s05rb.dut.cpp`

They do this by calling `sensorpath2008_brd_fast_spi_rd` from the `_general_access_rd` mandatory function.

```
if ((access_method == WV_DEBUG_ACCESS_MEMORY_MAPPED) &&
```

```
(address >= SENSORPATH_DATA_ADDR_BASE && address < (SENSORPATH_DATA_ADDR_BASE +
SENSORPATH_DATA_ADDR_SIZE))) {
    WvWord config_index = address - SENSORPATH_DATA_ADDR_BASE;

    if (config_index >= SP2008_DIFF_NUM_CAP_CONFIGS) {
        return script_process_return(
            INTERFACE_RET_ERROR,
            "Configuration index out of bounds"
        );
    }

    interface_ret = sensorpath2008_brd_fast_spi_rd(
        data->board_ptr,
        data->cap_config[config_index].Name,
        &rd_val
    );
}
```

The user of the DLL reads the SensorPath ADC channels by attempting to read a value from a designated area in the Cypress memory space. If the address does fall within this memory space, then `sensorpath2008_brd_fast_spi_rd` is called. For more details on the memory mapping, please consult `sensorpath.doc`.

## 9.8 *sensorpath2008\_none\_optical\_encoder.dut.cpp*

This device does not have a channel select option. This device will always output one 16-bit sample from channel 0 and one 16-bit sample from channel 1. There are two separate serial protocol arrays.

```
const unsigned char sp2008_opt_enc_fast_spi_1ch_cmd[SP2008_OPT_ENC_CMD_1CH_SZ] = {
```

This one reads 32 bits from the device.

```
const unsigned char sp2008_opt_enc_fast_spi_2ch_cmd[SP2008_OPT_ENC_CMD_2CH_SZ] = {
```

This one reads 64 bits from the device. The difference from the previous one is that this allows the fastest possible method to capture four samples at once. The first 32 bits will be the same as the previous array. But then, there are no pauses and another 32 bits are read back immediately. This second set of 32 bits is the next set of samples from the device. The ability to read 4 samples as fast as possible was a requirement from the GUI application.

The API used to acquire these samples is on a per sample basis. One memory access at the proper memory address returns only one sample. To keep all samples consistent, only configuration 0 and configuration 2 result in the serial protocol array being sent to the device.

In the other configurations, the previous values stored in the firmware return array is used.

So:

0xcf00 – configuration 0 – send 32-bit serial protocol array. Return CH0 16 bit sample.  
 0xcf01 – configuration 1 – use previous firmware return array. Return CH1 16 bit sample  
 0xcf02 – configuration 2 – send 64-bit serial protocol array. Return first CH0 16 bit sample  
 0xcf03 – configuration 3 – use previous firmware array. Return first CH1 16 bit sample  
 0xcf04 – configuration 4 – use previous firmware array. Return second CH0 16 bit sample

0xc05 – configuration 5 – use previous firmware array. Return second CH1 16 bit sample

## 9.9 *sensorpath2008\_none\_sp1202s01rb.dut.cpp*

This ADC does not require an explicit channel selection and so this is a simplified version of the serial protocol array in the board script. This simply reads 16 bits from the device.

## 9.10 *sensorpath2008\_none\_sp1602s01rb.dut.cpp*

This ADC also does not require an explicit channel selection. However, the number of clock cycles required to read the data is 18 clock cycles and so the serial protocol array is resized to accommodate this.

## 9.11 *Customizing a serial protocol array*

The simplest example to start with is the one from *sensorpath2008\_none\_sp1202s01rb.dut.cpp*. This simply reads data.

When constructing the array, please remember to allocate 3 bytes for the command header. By doing this, much of the code from the other scripts can be copied.

It's a good idea to keep a comment in the serial protocol array to keep track of the current index, just to ease the bookkeeping.

If the serial protocol requires the modification of the serial protocol array, then copy the following loop just before sending the serial protocol array to the firmware:

```
for (i = 0; i < SENSORPATH2008_BRD_ADDR_SZ; i++) {
    // Assume MSB first always
    new_bit = ch & (1 << (SENSORPATH2008_BRD_ADDR_SZ - i - 1));

    // Write twice for each cycle of the clock
    if (new_bit) {
        data->tx_buf[offset] |= mask;
        data->tx_buf[offset+1] |= mask;
    } else {
        data->tx_buf[offset] &= (~mask);
        data->tx_buf[offset+1] &= (~mask);
    }
    offset += 2;
}
```

The serial protocol array will need to be modified when a specific channel or a register address needs to be specified. The variable that contains the channel number of register address is *ch*.

The constants in the above loop will need to change according to the number of elements in the serial protocol array.

## 10 Debugging

Currently, the only way to debug the files is to look at the log files. So, the more debug output in the script, the better.

However, there are several steps in the validation process that will assist in isolating any problems.

Step 1: use `cint_test.exe`

Step 2: use WV5 GUI with a simulator

Step 3: use WV5 GUI with an actual board

### 10.1 *cint\_test.exe*

This is a utility created to test the C script. It attempts to run every single mandatory function. Also, if a GUI is defined, then the utility will attempt to force a change on every GUI element.

This script does not use the hardware so a board is not necessary; all of the low-level communication is simulated. However, the simulation is quite limited. For this reason, this utility should be used only as one step in the entire process of script validation; just because the script seems to pass this test does not make the script ready for use.

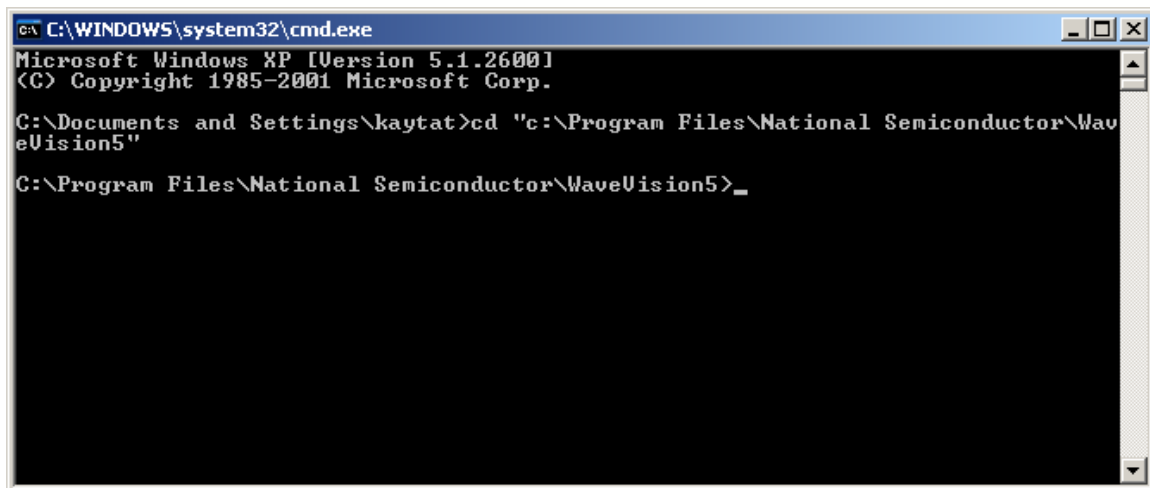
#### 10.1.1 *Location*

This utility should be located in "C:\Program Files\National Semiconductor\WaveVision5".

#### 10.1.2 *Running*

Open a command prompt by navigating to Start->Programs->Accessories->Command Prompt.

Change the directory to "C:\Program Files\National Semiconductor\WaveVision5"



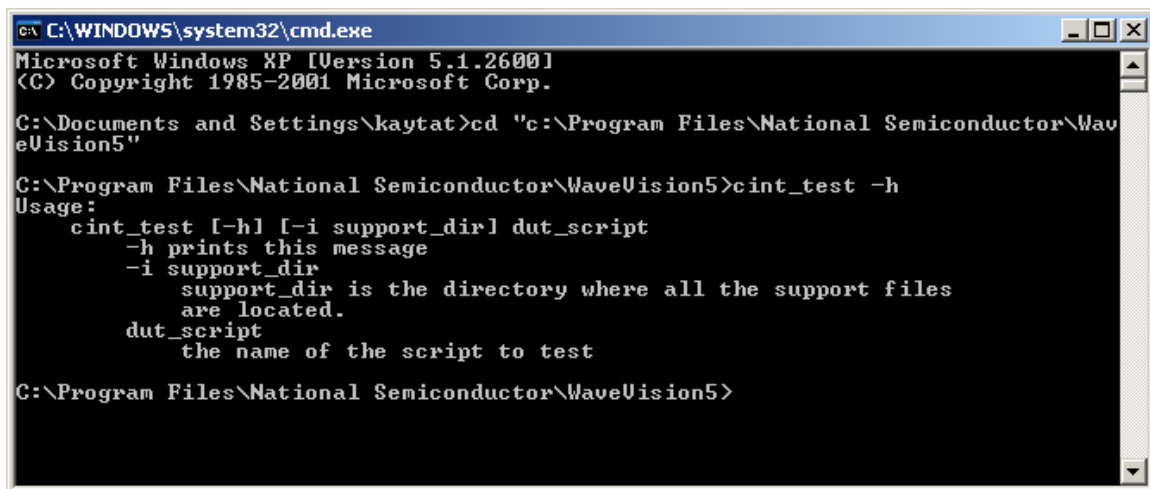
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\kaytat>cd "c:\Program Files\National Semiconductor\WaveVision5"

C:\Program Files\National Semiconductor\WaveVision5>
```

### 10.1.3 Command line options

Type "cint\_test -h". The "-h" is the help option.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\kaytat>cd "c:\Program Files\National Semiconductor\WaveVision5"

C:\Program Files\National Semiconductor\WaveVision5>cint_test -h
Usage:
  cint_test [-h] [-i support_dir] dut_script
  -h prints this message
  -i support_dir
      support_dir is the directory where all the support files
      are located.
  dut_script
      the name of the script to test

C:\Program Files\National Semiconductor\WaveVision5>
```

### 10.1.4 Testing the script

Type "cint\_test -i hardware wv5\_xc4vlx25\_adc14ds105.dut.cpp"

```

C:\WINDOWS\system32\cmd.exe - cint_test -i hardware wv5_xc4vlx25_adc14ds105.dut.cpp

C:\Documents and Settings\kaytat>cd "c:\Program Files\National Semiconductor\WaveVision5"

C:\Program Files\National Semiconductor\WaveVision5>cint_test -h
Usage:
  cint_test [-h] [-i support_dir] dut_script
    -h prints this message
    -i support_dir
        support_dir is the directory where all the support files
        are located.
    dut_script
        the name of the script to test

C:\Program Files\National Semiconductor\WaveVision5>cint_test -i hardware wv5_xc4vlx25_adc14ds105.dut.cpp

**** init_board ****
  
```

You will be prompted before executing each mandatory function. Hit the enter key to continue through the entire list of functions.

At the end, hit any key and then enter to exit the utility.

```

C:\WINDOWS\system32\cmd.exe

**** Testing poll ****

**** end_board ****

**** Hit any character and enter to exit ****
c
c
C:\Program Files\National Semiconductor\WaveVision5>
  
```

### 10.1.5 Files generated

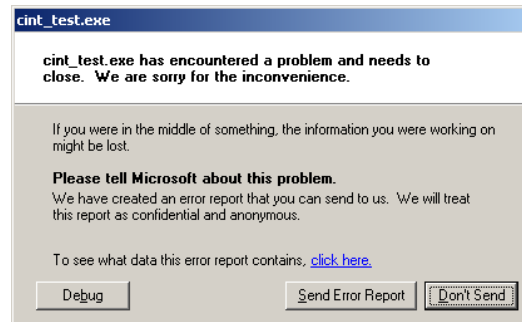
This utility will generate two log files in "C:\Program Files\National Semiconductor\WaveVision5." The first one is diag.txt and this contains the context of all the script function calls.

The second is exetrace.txt which lists all the lines the script executed. Unfortunately, this script is sometimes unreliable. The reason is this log file only gets updated periodically. This means if there was a fatal error in the script, then it is possible this log file will not actually show the offending line of code.

So, both of these two log files must be investigated to determine the context of the problem and the line of code that caused it.

### 10.1.6 Crash

If the utility crashes, then most likely the script encountered a fatal error or one of the return values is causing the DLL to crash. You will see a message like this.



To generate this error, the script was modified to force a null pointer in one of the tab data structures.

```
data->tabs[0].caption = 0;
```

This is the modified line

```
data->tabs[0].caption = "Hardware Register Tab";
```

This is the original line.

In this case, the DLL is crashing due to an invalid pointer in the `caption` field. Looking at `diag.txt`, the last few lines are:

```
09/04/24:11:04:43.0484:SCRIPT_EP:gui_tab_enum:
cmd:wv5_xc4v1x25_adc14ds105_gui_tab_enum(active_duts[0])
09/04/24:11:04:43.0484:SCRIPT:INTERFACE_RET_OK:wv5_xc4v1x25_adc14ds105_gui_tab_enum
09/04/24:11:04:43.0484:SCRIPT_EP:gui_tab_enum: tab count:2
```

Each line of the log file starts with a time and identifier for the subsystem that is generating the log entry.

`:SCRIPT_EP:` is the subsystem in the DLL that interacts with the scripts. `:SCRIPT:` is the log output generated by the script using the `script_debug_out_xyz` helper calls.

So in this case, `:SCRIPT:` is returning a success for the `_gui_tab_enum` mandatory function. And then the DLL is trying to continue but crashes because there are no other lines after `:SCRIPT_EP:`.

Although it looks like the DLL is crashing, the script is not off the hook. The best way to debug situations like this is to comment out entirely the script code and then slowly reintroduce the script code.

If the script code is reintroduced in its entirety and the app is still crashing, then it may be a DLL problem.

### 10.1.7 Junk in diag.txt

If the utility doesn't crash, then the first step is to look at diag.txt.

When there is a problem in the script, you might see some junk in the diag.txt that should really belong in exetrace.txt log file. This is an indication that there is some type of compilation error in the script file.

As an example, I purposely created an error in the script file and diag.txt contained these entries:

```
09/04/24:11:23:59.0406:SCRIPT_EP:version: cmd:wv5_xc4vlx25_adc14ds105_version()
09/04/24:11:23:59.0437:SCRIPT_EP:execute_dut_initialization:
cmd:wv5_xc4vlx25_adc14ds105_execute_dut_initialization(active_duts[0])
09/04/24:11:23:59.0437:SCRIPT_EP:!!!Dictionary position rewound...
09/04/24:11:23:59.0437:SCRIPT_EP:# wv5_xc4vlx25_adc14ds105.dut.cpp
09/04/24:11:23:59.0437:SCRIPT_EP:900
wv5_xc4vlx25_adc14ds105_execute_dut_initialization(Adc14ds105_Data* d data) {
09/04/24:11:23:59.0437:SCRIPT_EP:
09/04/24:11:23:59.0437:SCRIPT:wv5_xc4vlx25_adc14ds105_execute_dut_initialization
09/04/24:11:23:59.0437:SCRIPT_EP:901
script_debug_out_pre("wv5_xc4vlx25_adc14ds105_execute_dut_initialization");
09/04/24:11:23:59.0437:SCRIPT_EP:
09/04/24:11:23:59.0437:SCRIPT_EP:902      script_debug_out_endl();
09/04/24:11:23:59.0437:SCRIPT_EP:
09/04/24:11:23:59.0437:SCRIPT_EP:903
09/04/24:11:23:59.0437:SCRIPT_EP:
```

The last few lines of this snippet is showing lines that should really be in exetrace.txt. Taking a closer look, the DLL is attempting to call `wv5_xc4vlx25_adc14ds105_version()` but there is no corresponding return line from the script. This would indicate there may be something wrong with `wv5_xc4vlx25_adc14ds105_version()`.

Looking at exetrace.txt:

```
# wv5_xc4vlx25_adc14ds105.dut.cpp
157 wv5_xc4vlx25_adc14ds105_version() {

158      xscript_debug_out_pre("Version:%s", ADC14DS105_VERSION);Error: Function
xscript_debug_out_pre("Version:%s",ADC14DS105_VERSION) is not defined in current scope
wv5_xc4vlx25_adc14ds105.dut.cpp(158)

!!! 0 object(s) deleted by Reference Count Control !!!
```

Here, you can see that the scripting engine is complaining about `xscript_debug_out_pre`.

And sure enough, the error that was artificially injected was a typo for the debug call.

```
xscript_debug_out_pre("Version:%s", ADC14DS105_VERSION);
```

This line of code was replaced with:

```
script_debug_out_pre("Version:%s", ADC14DS105_VERSION);
```



### 10.1.8 Failures to ignore

This utility attempts to exercise all mandatory functions. The boards that do not support this functionality will generate an error code in diag.txt.

The tests that can be ignored include:

```
get_dac_download_parameters
```

Reason: ADC14DS105 is not a DAC.

## 10.2 Using a simulator for the WV5 GUI

For the initial validation of the script, it is useful to be able to test the GUI without the use of hardware so a simulator was created. *The WV5 Simulator is described in detail in the WV5 System Designers' Guide. Please refer to that document before proceeding with this section.*

Due to the changes in the directory structure, the \_\_sim.xml file should be placed in hardware\scripts and not the firmware directory.

Since the example script is for the ADC14DS105, \_\_sim.xml file should contain:

```
<wv5>
  <sim vid="0x0400" pid="0x2007" dut_desc="adc14ds105" data_params="13 0 29 16"/>
</wv5>
```

To enable the simulator, make sure to copy \_\_sim.xml to hardware\scripts and make sure there is no hardware plugged into the PC.

The process for debugging is the same as with cint\_test.exe, namely the log files must be consulted to see where the failure occurred. However, in this case, diag.txt is not used and instead Data\LogFiles\DLLLog.txt should be consulted.

### 10.2.1 Simulator Limitations

The simulator does not provide an exact functional simulation of the device. The simulation is fairly brain-dead. In fact, at the lowest level, all it does is throw away USB transmit requests and generates a random byte for every USB receive request.

This will have consequences when a register is read-back in the script. For instance, the level 1 register read will attempt to read the hardware register back. In the simulation environment, the simulator will basically return a randomized piece of data.

This may confuse the script because the simulator may return values in register fields that should be all zeros in the physical hardware.

Please be aware of this limitation. To work around this, the script is advised to always sanitize the return values.

As an example of how to do this, please see the code in the example script that handles the change for tab 1. Tab 1 contains one level 1 register element. And in that example, the value read back from the hardware is cleaned up before being assigned to the GUI.

### *10.2.2 Data\LogFiles\DLLLog.txt*

This log file contains the log from all of the DLL and so it has more information than just script output. This log file should be consulted to give context to when the failure occurred.

### *10.2.3 exetrace.txt*

This log file is the same as the cint\_test.exe case. This log file will contain all the code executed by the script.

## *10.3 Using a real board with the WV5 GUI*

This is the final step in the validation. The process is the same; check the logs if there are any problems.