XMOS Specifications and Testplans

Version 0.1

XMOS

This document describes how to use ReStructedText with some XMOS extensions to create specification and testplan documents.

1 Overview

1.1 Specification and Features

A specification document describes a product - usually a software reference design, component or library. The document should contain a complete description of the product such that it provides an exact description to base an implementation on and a document that can be reviewed by marketing/customers/field engineers to adjudge the specific implementation with respect to customer need.

The specification should include several elements including description of scope, a precise list of features, limitations, resource usage etc. The full details of these requirements for a specification can be found at http://cognidox/cgi-perl/part-details?partnum=XM-000059-PR. This document does not cover this but describes the method of formally documenting the feature list.

The feature list is a key component of the spec. In a our formal specification documents each feature is uniquely identified by a label, conventionally made up of capitials seperated with underscores. Here is an example for the UART specification of a labelled feature regarding the buffering in the component:

UART_RX_BUFFERING:

> The UART RX component will buffer data received over the serial link and queue the data to be sent to the client. In the case where the buffer is full incoming bytes of data will be dropped.
>
> **Parent Feature** UART_RX

Note that a feature can have a parent feature it belongs to. So the features form a heirachy. There is no stipulation on this heirachy (or even using one) but some product feature sets will naturally fall into this format which can aid reporting later on.

Generally, the feature list is just a list of these labelled features with their descriptions. The only other important consideration is that some features can be configured in a number of disjoint ways. It is important to capture these in the specification so that when the product is validated all the configurations are tested. Here is an example of a feature with multiple configurations:

CODEC_SUPPORT:

The software will be capable of interfacing with a CODEC over I2S. The device will be capable of interfacing to the codec in slave or master mode.

config_options MASTER | SLAVE (default)

Note that the specification also specifies the default configuration for the feature (there must always be a default).

1.2 Tests and Test Plans

Test plan documents fundamentally provide a list of tests to peform to validate the functionality of a product. The test list specifies which features are tested by each test to determine coverage.

The testplan document specifies a list of *tests* but each test may need to be run for several configurations/parameters. So the test list expands into a list of *test instances*.

For example the following test specification:

midi_loopback_windows

1. Connect gameport on DUT to MIDI gameport cable connected to Machine B
2. On machine B. Run MIDI OX. To set up a loopback open Options->MIDI Devices. Select E-DSP in MIDI Inputs and MIDI Outputs. Click OK.
3. On machine A. Run miditest. Select MME driver, XMOS interface for input and output and ensure all message options except Midi tick are enabled.
4. Click Test - this will alert the tester to any errors in the data sequence which will result in a failure.

configurations OS_WIN.*

features BUILD_OPTION_MIDI, MASTER_CLOCK_DIVIDER_12_22

setup windows_platform

:test_time : 10

Will expand into several tests for different windows platforms:

```
midi_loopback_windows_OS_WIN7
midi_loopback_windows_OS_WINXP
midi_loopback_windows_OS_WINVISTA
```

The expansion can be got from the testplan and specification using the **validate_testplan.pl** script. The test database also expands the test to create test runs for validation.

1.3 ReStructuredText

Both specifications and testplans are created using ReStructuredText. This is a markup language for creating documents. It is part of the python docutils package. For more details see:

http://docutils.sourceforge.net/

In addition to the standard rst markup. We have added several *Custom Directives* for describing features and tests in documents. These extra directives allow tools to automatically read in specifications and testplans for validation.

1.4 Important Tip

Indentation is very important in ReStructuredText. In particular, always ensure that the body of the directive is indented identically to its options. For example this will produce an error:

```
.. feature:: SPDIF_RX
     :summarize:
     :config_options: ON (default) | OFF

   S/PDIF transmit can be enable or disabled as a build option
```

To fix it it needs to be changed so the body text aligns with the options like this:

```
.. feature:: SPDIF_RX
     :summarize:
     :config_options: ON (default) | OFF

     S/PDIF transmit can be enable or disabled as a build option::
```

2 Custom Directives

The following extensions are added to rst to specify features and tests in specification and testplan documents. They follow the usual syntax for directives:

```
.. directive:: [directive argument]
   :option1: [option args]
   :option2: [option args]

   Directive body
```

2.1 Features

.. feature:: `name`

This directive describes a feature. Its one argument is the feature name (following the convention of a capitalized identifier with underscores for spaces).

Example:

```
.. feature:: SPDIF_RX
   :summarize:
   :config_options: ON (default) | OFF

   S/PDIF transmit can be enable or disabled as a build option.
```

The directive has the following options:

summarize
: This option affects whether this feature will be part of the top level summary of any validation run

config_options
: This option specifies the configurations for a feature.

runtime
: This options says that the configurations of the feature are runtime options (not build options)

summarize_options
: If this option is specified then the configurations of the feature are summarizes in testing reports

2.2 Tests

.. test:: `name`

This directive specifies a single test (which will be expanded into multiple instances). The name argument is the identifier for the test and generally follows the same format as features (capitalized seperated with underscores)

Example:

```
.. test:: hw_loopback_test
   :setup: XC1A_LOOPBACK
   :configurations: STOP_BITS_.*, PARITY_BITS_.*, BITS_PER_BYTE_.*
   :features: UART_RX, UART_TX
   :test_time: 2

   This test runs 2 uart tx/rx loopbacks. The UART TX component is used to
   output data which is received by UART RX component.
```

The directive has the following options:

features

This option takes an argument which is the comma separated list of features that the test covers.

configurations

This option takes an argument which specifies which configurations to make test instances of this test for. The format of this argument is as follows:

```
:configurations:  pattern1, pattern2, ...
```

where a pattern is of the form:

```
regexp [excluding regexp]
```

A pattern specifies a set of config options for a particular feature. For example if we specifiy a feature:

```
.. feature:: PARITY_BITS
     :config_options: none (default) | even | odd
```

Then the pattern:

```
PARITY_BITS_.*
```

Will match all configurations of parity bits. The match is done against the feature name postfixed with an underscore and the option.

The cross products of the list of patterns in the configuration option gives the entire configuration space under which the test will be performed. For example the following:

```
:configurations: PARITY_BITS_.*, STOP_BITS_.*
```

Will cause the test to expand for all combinations of parity bits and stop bits.

If a config option for a feature is not mentioned in the test then the default is assumed.

parameters

This option takes an argument which specifies the parameters of the test. The format of this argument is as follows:

```
:parameters: param1 = option1 | option2 | ... , param2 = option1, option2, ...
```

For example:

```
:parameters: sample_rate = 192 | 88.2 | 48 | 176.4 | 96 | 44.1,  driver = asio,
```

This specifies two parameters (sample_rate and driver) and their options. The test will be expands across the entire combined space of parameters and configurations. So in the above example there will be a test for each configuration at each sample rate with each driver.

test_procedure

This option takes an argument which describes a common test procedure that is required to take the test, if one exists.

test_time

This option takes an argument which describes the time it takes to run the test (not including the setup time) in minutes.

priority

This options describes the priority of the test. It takes a single integer argument (lower is higher priority).

.. test_procedure:: `name`

This directive describes a test procedure that could be common to many tests.

Example:

```
.. test_procedure:: windows_7_input
  :setup: windows_platform
  :direction: input

  #. Access P:\applications\AudioTest\TestingTools
  #. Ensure no other scripts or applications are running
  #. Launch the shortcut titled "Run Windows L2 Input Tests"
  #. Click "Start Tests"
  #. Follow the onscreen instructions for completing the tests
  #. Once all tests are complete drag the directory containing all the session f
```

.. setup:: `name`

This directive describes a phyiscal test setup needed to run a test.

Example:

```
.. setup:: windows_platform
 :setup_time: 15

 Three version: Windows 7, Windows Vista and Windows XP

 This setup is for the validation of the Thesycon Driver

 #. Boot the required windows platform
 #. Retrieve the appropriate binary from the RC binary set
 #. Connect the USB cable and XTAG2 to the DUT
 #. Utilise XFLASH to load the firmware to the board
 #. Install driver onto the test machine, if the driver is already installed the
 #. Remove any previous instance of the XMOS Audio solution using the USBDeview
 #. Connect the DUT
 #. Ensure device is shown as connected in the Thesycon Control Panel
 #. Ensure that all system volumes are set to maximum
```

The directive has the following options:

setup_time
: This option has one argument which is the time taken to do the setup in minutes.

3 How to Write a Spec and Testplan: An Example

This section explains how to use the directives specified in the previous section to construct meaningful testplans for XMOS software components.

3.1 Define the System Under Test

The system under test is defined by creating a feature hierachy or feature list:

```
.. feature:: UART_RX

    A UART Rx component will allow the client to send data over a
    serial link.

.. feature:: BUILD_OPTION_UART_RX_BUFFER_SIZE
   :parents: UART_RX

   A ``#define`` will set the size of the UART Rx component buffer.
```

A typical component may well have multiple build options. This means that it is configured at compile time into one mode or another. Some features from the list will apply in both modes, and some will only apply in one of the modes.

For example, the uart is configured at compile time for a specific baud rate. So to specify this, first we define the feature that may be configured, and then specify the options available:

```
.. feature:: BAUD_RATE
  :config_options: 24K | 96K | 115.2K
  :parents: UART_TX, UART_RX
```

In the case above, the baud rate feature is in fact settable at any value between 1 and 115200, but for the testplan we only plan to test at three baud rates.

In the case below, there is an option to compile in a specific behaviour, and the possible values are an enumeration (in this case with 2 values, on and off):

```
.. feature:: BUILD_OPTION_UART_RX_EXCEPT_ON_OVERFLOW
   :config_options: OFF | ON
   :parents: UART_RX

   There will be a build option to raise an exception on RX buffer
   overflow for debugging purposes.
```

3.2 Define the Test Setup

Once the system under test is defined, define the various methods by which it is to be tested. For example, a typical component will have a simulation testbench, a hardware testbench and possibly a hardware demo.

We can define these like this:

```
.. setup:: SIMULATOR
  :setup_time: 1

  Setup a host PC with the latest released development tools.

.. setup:: XC1A
  :setup_time: 2

  Setup a host PC running windows with the latest development tools and
  an XC-1A dev card attached.
```

3.3 Define the Tests

Once the feature tree is completed and all the different testing setups, we can define the tests, with each test referencing some of the features and a specific test setup:

```
.. test:: sim_loopback_test
  :setup: SIMULATOR
  :configurations: BAUD_RATE_.*, BITS_PER_BYTE.* , PARITY_BITS.*, STOP_BITS_.*
  :features: UART_RX, UART_TX
  :test_time: 5
```

This test definition is going to expand into 72 tests:

(3 baud rates) x (4 bits_per_byte settings) x (3 parity bit settings) x (2 stop bit settings)

Note that different tests in this set must be run using different test binaries of the firmware because the options being tested are all compile time options of the component.

3.4 Define Summary Options

A method for summarizing testing in progress can be defined by using the :summarize: and :summarize_options: with the feature directive.

This will serve to catagorise tests as being required to complete the verification of a given catgory. For example:

```
.. feature:: BAUD_RATE
  :summarize_options:
  :config_options: 24K | 96K | 115.2K
  :parents: UART_TX, UART_RX

.. feature:: PARITY_BITS
  :summarize_options:
  :config_options: none | even | odd
  :runtime:
  :parents: UART_TX, UART_RX
```

The above will create separate summary items for BAUD_RATE and PARITY_BITS. Sample output from the testplan validation tool described below for the directives baove might be something like:

```
BAUD_RATE_24K: 24 tests
BAUD_RATE_96K: 24 tests
BAUD_RATE_115.2K: 24 tests
PARITY_BITS_none: 32 tests
PARITY_BITS_even: 32 tests
PARITY_BITS_odd: 32 tests
```

For large testplans which take some time to develop and get all the tests passing, this is a useful way to quickly report progress against speicifc key features. Note that with this method, a single test can belong to multiple summary items.

3.5 Validate the Testplan

Once the testplan is completed it should be validated using the tools supplied by XMOS. The tool is run as follows:

```
validate_testplan <testplan.rst> <spec.rst>
```

The tool will report inconsistencies - for example a test that refers to a feature which is not defined.

To output the summary table, the -s option can be used:

```
validate_testplan -s <testplan.rst> <spec.rst>
```

To output an exppanded list of all tests, use the -l option:

```
validate_testplan -l <testplan.rst> <spec.rst>
```