
Creating DSP Conductor™ Primitives, a Tutorial

by Jon Hilbert
Cirrus Logic, Inc.

1. Introduction

This document describes how to write assembly code and XML files to implement a new DSP Conductor™ primitive algorithm. DSP Conductor primitives are devices that can be dragged onto a design canvas and wired into the signal flow of the target DSP device.

Primitives are defined within the DSP Conductor framework by three XML files and one or more object (.O) files. The XML files are:

- ***device_name.if.xml*** - specifies design-time attributes of the device, as well as definitions of run-time controls.
- ***device_name.pres.xml*** - specifies graphic information about how the DSP Conductor GUI should display the device.
- ***device_name.imp.xml*** - specifies information about data structures and code required to incorporate the device into a design, so that DSP Conductor can compile the device into the final loadable file.

Typical DSP Conductor XML files are fairly small and define only those aspects of a primitive unique to that primitive. An "include" capability factors the common aspects of the definition XML to common *include* files. All three XML files must be named with a common prefix (the *device_name* shown above). For example, the files that implement the *Gain* primitive are called: *gain.pres.xml*, *gain.if.xml*, and *gain.imp.xml*.

The object files implement the digital signal processing for the primitive. The following discussion presumes that the digital signal processing code is written in assembly language. Discussion will draw upon implementation of the Cirrus-supplied *Gain* device for examples. For most primitives, a single object file is sufficient, and it is named *device_name.O*.

2. Terms and Definitions

The following terms are commonly used in this document:

Audio buffer - A buffer allocated by DSP Conductor to hold Bricks of audio data flowing from one primitive to another over a wire specified in the project design schematic. In a DSP Conductor design, each wire is represented by a buffer allocated in Y memory on the DSP. DSP Conductor automatically reuses these buffers wherever possible to reduce demand for on-chip memory.

Brick - A vector of PCM audio samples. For efficiency, audio data is processed by DSP Conductor primitives in vectors.

Brick Size - Length of the PCM audio vector in audio samples. Brick Size is typically 16 samples, although it can be changed by a project design property.

Control - A control is a GUI that allows manipulation of one or more public data variables for a primitive. The control may be a write control, or a read control (but not both). In addition to the GUI representation, a control is represented in the public data structure for the algorithm. Definitions in the interface and implementation XML files for the primitive tie the GUI to the public data value.

Control panel - The aggregate GUI representation of all of the individual controls for a primitive. The presentation XML file allows you to define the layout of the control panel and many other aspects of the appearance of the controls.

Device - An instance of a primitive when it has been added to a project design schematic.

Microkernel - The Cirrus-supplied firmware that provides an operating environment for DSP Conductor designs to run in. The microkernel provides algorithm scheduling and parameters for the defined entry points of each primitive. The microkernel also manages all hardware access for audio data I/O. The microkernel may also provide commonly useful routines that may be accessed by algorithm code.

Port - A representation of an access point for data flowing into or out of a primitive in a DSP Conductor design schematic. In the design canvas, ports for audio data look like connectors at the left or right edge of a primitive block, while ports for logic signals look like connectors at the top or bottom edge. Ports also have representations in the public and/or private data structures for an algorithm, where they are implemented as indirect pointers to data buffers.

Primitive - The collection of object and XML files that define an audio processing algorithm that can be included in a DSP Conductor design. A primitive is displayed as a single-colored block in a DSP Conductor design, and cannot be decomposed into lower-level primitives.

Private data - Data structure that is used by an instance of an algorithm that contains variables that are used for storing internal state on the DSP, not accessible to outside processes at all. This data structure is produced by DSP Conductor based on definitions in the implementation XML file.

Property - Variable set in the DSP Conductor designer before a project is compiled and deployed. There are project properties and device properties. Although device properties might be used as default control values, Property values cannot be changed at runtime through controls,.

Public read-only data - Data structure that is used by an instance of an algorithm that contains variables that are intended to be read (but not written) by outside processes. This data structure is produced by DSP Conductor based on definitions in the implementation XML file.

Public read/write data - Data structure that is used by an instance of an algorithm that contains variables that are intended to be manipulated by outside processes, either a host microprocessor (or something else in the CobraNet™ environment). This data structure is produced by DSP Conductor based on definitions in the implementation XML file.

Public data - The union of public read/write and public read-only data spaces.

Ruid - Within the primitive XML files, a RUID is a "relatively unique id"; e.g., a name that, within the context of the XML file, is unique. Ruids are used to identify port names, control names, and device types.

Static data - Data structure shared by all instances of an algorithm. This data is defined in the algorithm code file(s) and is not manipulated or created by DSP Conductor at all.

Stride - The size of the increment of an I/O buffer pointer as used in a primitive to step through the buffer. I/O buffers may have different interleaving factors, and so for each buffer pointer used by a primitive it should also have a *stride* value that specifies how data is interleaved in the buffer. DSP Conductor defines two stride values that are used for all I/O buffers in a project.

3. Development Methodology

3.1 Start with something similar

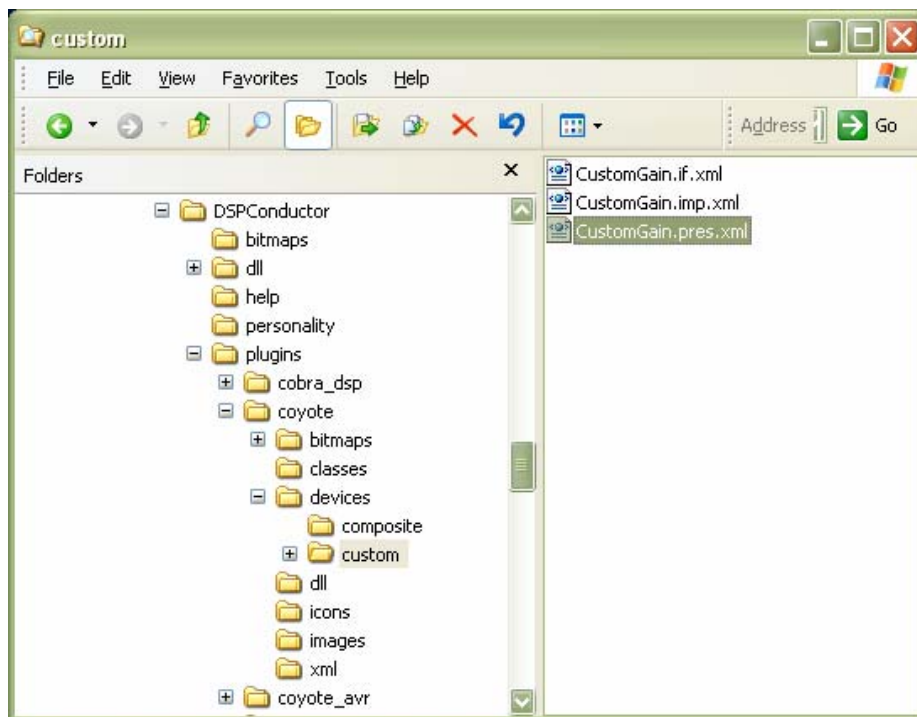
The easiest way to create a new primitive for DSP Conductor is to start by cloning the files associated with an existing primitive that is similar to the new one. Similarity is best determined by similarity of external interface qualities, rather than similarity of audio processing function. In other words, find a primitive whose audio port definitions and controls are most similar to those envisioned for the new primitive. Copy the *.xml files for the model primitive to the `\plugins\coyote\devices\custom` folder, as described below. After copying them, rename the files so that the base name of each file is something descriptive of the new primitive.

Important: Do not use blanks in the names of any of the XML files.

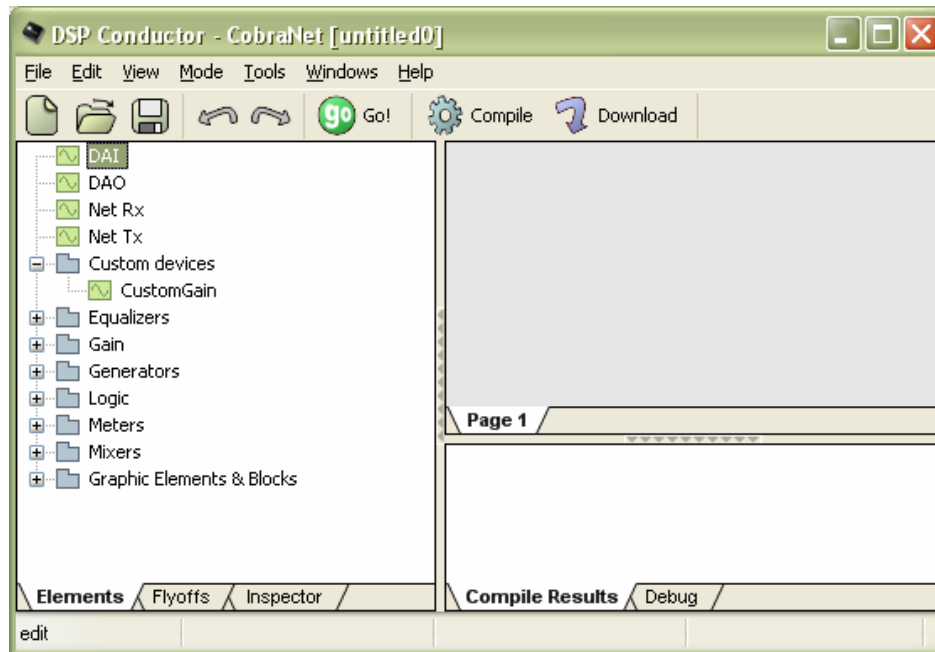
3.2 Adding to the device palette

To add a new device to the DSP Conductor menu, simply place all the relevant .xml and .O files in the `CoyoteCAD\plugins\coyote\devices\custom` folder and restart DSP Conductor. DSP Conductor will automatically add a menu item for the new device in the device palette, under *Custom Devices*.

For example, consider the following file/directory structure:



The directory contents shown above produce the following DSP Conductor device palette:



3.3 Edit-build-test Cycle

The best way to get a new primitive in working order is to first get the XML files properly defined, then get the primitive code working. There is, of course, some interaction between the XML files and the algorithm code, since the implementation XML file defines the data structure that will be made available to the algorithm at runtime, so it is a good idea to have some idea of the data required for an algorithm before getting started. The following flowchart seen in Figure 1. outlines a useful process for developing a DSP Conductor primitive.

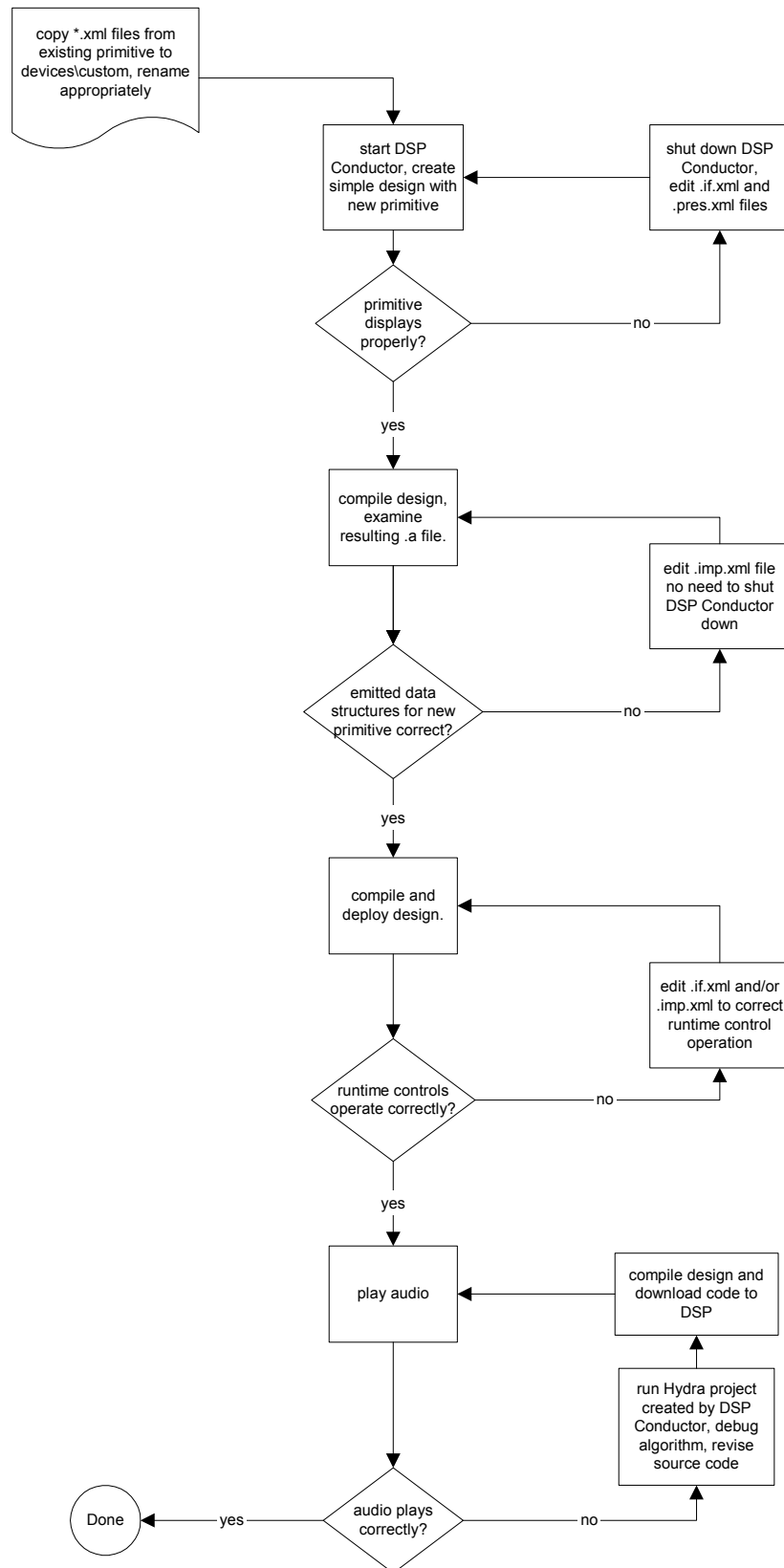


Figure 1. Process for Developing a DSP Conductor Primitive

4. Primitive XML Files

4.1 XML File Syntax

The three XML file types that are used to define a primitive are described in separate sections below. There are, however, some common aspects of the XML language used in the three files:

□ **<xpp:>**

DSP Conductor XML files use elements in the *xpp:* namespace to denote items that are to be processed by the XML preprocessor in DSP Conductor. The XML preprocessor can handle several different syntactic constructs, the ones that most often occur in primitive XML files are:

- **<xpp:include file="name of file">**

This construct includes the specified file into the current XML file. This is semantically analogous to the C preprocessor *#include* construct.

- **<xpp:define name="definition name">**

This construct defines a set of elements and associates them with *<definition name>*. The *<definition name>* can then be referenced in later XML with the effect that all of the sub-elements specified in the *<xpp:define>* element are copied into the XML at the point of reference. The same *<definition name>* can be defined multiple times within an XML file, with only the latest definition being used in any subsequent *<xpp:insert>* of the definition.

- **<xpp:insert name="definition name">**

This construct is a reference to an earlier *<xpp:define>* construct. The sub-elements of the *<xpp:define>* construct are inserted into the XML at the point where the *<xpp:insert>* element occurs, replacing it.

- **<xpp:repeat index="variable" first="expression" last="expression">**

This looping construct repeats the contained elements for (last-first+1) times, giving the index variable the current value. Normally the index variable itself is used within the contained elements, although that is not required.

- **<xpp:evaluate expression="expression">**

Evaluates the algebraic expression. The most common use is to set a variable to some calculated value. The variable is then available for later use by other *<xpp:>* elements.

The main use of the following xpp constructs is to support the factoring of common aspects of primitive XML definitions to allow the definitions of different primitives to have common look, feel, and behavior.

□ **<xpr:element>expression</xpr:element> or xpr:attribute="expression"**

This element or attribute prefix indicates that the element or attribute may contain algebraic expressions which are to be evaluated.

□ **<sxpr:element>expression</sxpr:element>**

This construct is used in the implementation XML files only. It denotes an algebraic expression that is evaluated during XML processing. Examples of its use are given in the section on implementation XML files.

The following idiom is widely used in DSP Conductor XML files for factoring and parameterizing the common XML:

- a) Use `<xpp:define>` to define an empty set of override parameters; e.g.,

```
<xpp:define name="structure_x_overrides" />
```

- b) Use `<xpp:define>` to define a default set of values, with a call to insert the overrides; e.g.,

```
<xpp:define name="structure_x">
  ...
  <xpp:insert name="structure_x_overrides"
</xpp:define>
```

- c) When ready to use the default structure with instance-specific parameters, redefine the override parameters immediately before inserting the structure definition; e.g.,

```
<xpp:define name="structure_x_overrides"
  ...
</xpp:define>
<xpp:insert name="structure_x" />
```

In each of the following sections, a brief overview of the purpose and organization of the XML file is presented, followed by an example, a detailed analysis of the example, and finally by detailed description of the XML elements used in that type of XML file.

4.2 Presentation XML File (*device_name.pres.xml*)

This file defines the appearance of the GUI representation of the device including input/output port types, count and placement, controls, color values, and a default text description for the device. The sections and contents of the presentation file are described in the sections following the example.

4.2.1 Example: *gain.pres.xml*

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2005 Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<ningaloo_document xmlns:xpp="www.peakaudio.com/xmlns/xmlpreprocessor"
  xmlns:xpr="www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator">
  <device_type_presentation>
    <!-- device type identifier - the .pres.xml, .if.xml, and .imp.xml
      files must all have the same values for these elements -->
    <device_type_identifier>
      <family value="coyote_dsp"/>
      <type value="gain"/>
    </device_type_identifier>
    <!-- include commonly used definitions for device and runtime
      control presentation -->
    <xpp:include file="coyote_dsp_device.presinc.xml"/>
    <xpp:include file="ctl_args_stock.presinc.xml" />
    <xpp:include file="ctl_styles.presinc.xml" />
    <!-- device args - this section defines the basic appearance of device -->
    <xpp:insert name='device_args_level' /> <!-- this is a "level" device -->
    <!-- control panel - this section defines the layout of the
      runtime controls for the device -->
    <xpp:define name="control_panel">
      <column>
        <xpp:insert name="ctl_args_mute" />
        <xpp:insert name="ctl_button_wide" />
        <xpp:insert name="ctl_args_gain" />
        <xpp:insert name="ctl_knob_with_textbox_and_label" />
      </column>
    </xpp:define>
    <!-- device layout - this section defines the layout of the GUI components
      of the device, including audio and control signal ports -->
    <xpp:define name="device_layout">
      <!-- define input control signals -->
      <xpp:define name="device_input_control_signals">
        <xpp:evaluate expression="input_control_signals='&quot;enable&quot;'" />
      </xpp:define>
      <!-- and 1 x 1 audio in/out -->
      <xpp:insert name="coyote_dsp_1_by_1_presentation"/>
    </xpp:define>
    <xpp:insert name="device_instantiation"/>
  </device_type_presentation>
</ningaloo_document>
```

4.2.2 Boilerplate

Common boilerplate for all .pres.xml files appears from the beginning of the file through the `<device_type_identifier>` element. This boilerplate content should be reproduced exactly in each presentation file.

<device_type_identifier> - This section gives a unique name and version information to the primitive. The `<device_type_identifier>` section must be identical in all three of the XML files for a given primitive. The name used is also the root file name for all XML files associated with the primitive.

The sub-elements of `<device_type_identifier>` are as follows:

<family> - Under DSP Conductor this should always have the value = "coyote_dsp"

<type> - The value of this element uniquely identifies the primitive. This value should not contain spaces, and should be the same as the root filename of all of the *.xml files for the primitive being defined.

4.2.3 Standard includes

A set of common `<xpp:include>` elements that should always be present in a .pres.xml file. The included files contain the common definitions referenced in later sections of the file.

4.2.4 Device args (Arguments)

This section defines the color and basic size of a primitive in the GUI. The device args section is a simple `<xpp:insert>` of a pre-existing device class definition. Choose the class that most closely matches the new primitive and include its device args class. The existing classes are:

<code>device_args_controller</code>	Control Signal Device
<code>device_args_delay</code>	Audio Delay
<code>device_args_dynamics</code>	Audio Dynamics Processor (i.e., compressor, limiter, etc.)
<code>device_args_filter</code>	Audio Filter
<code>device_args_level</code>	Audio Gain Controller
<code>device_args_mixer</code>	Channel Mixer
<code>device_args_meter</code>	Signal Level Meter
<code>device_args_processor</code>	Complex Audio Processor
<code>device_args_generator</code>	Signal Generator
<code>device_args_detector</code>	Signal Detector
<code>device_args_input</code>	Input from External Audio Connection
<code>device_args_output</code>	Output to External Audio Connection
<code>device_args_router</code>	Channel Selector

4.2.5 Control Panel

This section defines the layout and components of any runtime controls for the primitive. The main layout is defined by `<row>` and `<column>` elements, each of which has attributes which further refine the presentation. Within the defined `<row>` and `<column>` sections are the definitions of the actual runtime control widgets. A great deal of flexibility is available here, but the vast majority of primitive runtime controls simply include instances of appropriate pre-defined control types, using the `<xpp:insert>` mechanism.

In the *gain.pres.xml* file example, the controls are contained in a single column. There are two controls in this control panel. Each control is represented by two `<xpp:insert>` elements. The first element for each control specifies parameter values used by the second element. The second element is an insert of one of the common control type definitions. The common control styles available are:

<code>ctl_fader</code>	A slider-like fader control
<code>ctl_fader_with_textbox</code>	A fader with a textbox showing its current value
<code>ctl_button</code>	A pushbutton (2 values) control
<code>ctl_button_wide</code>	A wide pushbutton control. The width is consistent with the width of knobs and faders.
<code>ctl_button_wide_with_label_left</code>	A wide pushbutton control with a label to the left
<code>ctl_button_enable</code>	A pushbutton control, enabled when pressed
<code>ctl_button_enable_wide</code>	A wide pushbutton control, enabled when pressed
<code>ctl_button_disable</code>	A pushbutton control, disabled when pressed
<code>ctl_button_disable_wide</code>	A wide pushbutton control, disabled when pressed
<code>ctl_button_mute</code>	A pushbutton control, mute when pressed
<code>ctl_button_mute_wide</code>	A wide pushbutton control, mute when pressed
<code>ctl_knob</code>	Round control knob with a scalar value
<code>ctl_knob_with_textbox</code>	Round control knob with a scalar value and numerical display
<code>ctl_knob_with_textbox_and_label</code>	Round control knob with a scalar value and numerical display and label
<code>ctl_knob_with_textbox_and_label_left</code>	Round control knob with a scalar value and numerical display and label to the left
<code>ctl_meter_level</code>	Displays an audio level in a vertical bar.
<code>ctl_meter_gain</code>	Displays a gain in a vertical bar.
<code>ctl_meter_reduction</code>	Displays a gain reduction in a downwards facing vertical bar.
<code>ctl_led</code>	A binary indicator
<code>ctl_led_with_label_left</code>	A binary indicator with label to the left
<code>ctl_combobox</code>	Displays text value of control and a dropdown set of choices when the user clicks on it.

<code>ctl_combobox_with_label</code>	Displays text value of control and a dropdown set of choices when the user clicks on it. Includes a label.
<code>ctl_combobox_with_label_left</code>	Displays text value of control and a dropdown set of choices when the user clicks on it. Includes a label to the left.
<code>ctl_textbox</code>	Displays text value of control.

4.2.6 Device Layout

The `<device_layout>` section defines the arrangement and number of audio and control ports for the primitive. In the GUI representation of the primitive, ports are connection points for wires carrying audio or control data. In the gain device example, there is a single audio input port, one audio output port, and a single control port called "enable". Since single input/output configurations are quite common, there is a predefined definition for this port layout.

Audio Port Definition

The predefined port configurations are:

<code>coyote_dsp_x_by_0_presentation</code>	Variable number of inputs; no outputs
<code>coyote_dsp_x_by_1_presentation</code>	Variable number of inputs; one output
<code>coyote_dsp_x_by_y_presentation</code>	Variable number of inputs; variable number of outputs
<code>coyote_dsp_1_by_0_presentation</code>	One input; no outputs
<code>coyote_dsp_1_by_1_presentation</code>	One input; one output
<code>coyote_dsp_1_by_y_presentation</code>	One input; variable number of outputs
<code>coyote_dsp_0_by_0_presentation</code>	No inputs; no outputs
<code>coyote_dsp_0_by_1_presentation</code>	No inputs; one output
<code>coyote_dsp_0_by_y_presentation</code>	No inputs; variable number of outputs

When a variable number of inputs presentation is used, the interface XML file (see below) must define a device property called `input_count` to indicate the actual number of input ports. Similarly, for variable number of output ports, the interface XML must define a device property called `output_count` to indicate the actual number of output ports.

Control Signal Port Definition

Control signals are 32-bit scalar values that are produced by some primitives and consumed by others. The values are often Boolean (0 or 1) but may be any 32-bit value. Control ports are specified in the presentation XML file by using the `<xpp:define>` construct.

```
<xpp:define name="device_input_control_signals">
  <xpp:evaluate expression="input_control_signals='<comma-separated-list>' " />
</xpp:define>
or
<xpp:define name="device_output_control_signals">
  <xpp:evaluate expression="output_control_signals='<comma-separated-list>' " />
</xpp:define>
```

where the *<comma-separated-list>* is a list of quoted port names in the form *"<name>"*;

For example, a device with two output control signals named 'mean' and 'clip' would have the following control signal port definition:

```
<xpp:define name="device_output_control_signals">
  <xpp:evaluate expression="output_control_signals='&quot;mean&quot;;&quot;clip&quot;' " />
</xpp:define>
```

4.2.7 Device Instantiation

The final element in the presentation XML file is:

```
<xpp:insert name="device_instantiation"/>
```

This element includes the common presentation definition for all DSP Conductor primitives.

4.3 Interface XML File (.if.xml)

This file defines how the device is named in the GUI, what its design-time device properties are, what runtime control values it has, and its connections (ports). The information about runtime controls and ports is complementary to the information described in the presentation XML file. For runtime controls, the presentation XML file describes how they look in the GUI; the interface XML file describes what kind of values they can take on. For ports, the presentation XML file describes how they look in the GUI; the interface XML file describes what kind of data they take, assigns names and RUIDs.

The sections and contents of the interface file are described in the sections following the example shown in its entirety on page 14.

4.3.1 Example: gain.if.xml

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2005 Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<ningaloo_document xmlns:xpp=www.peakaudio.com/xmlns/xmlpreprocessor
  xmlns:xpr=www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator>
  <device_type_interface>
    <!--TYPE INFORMATION-->
    <device_type_info>
      <device_type_identifier>
        <family value="coyote_dsp"/>
        <type value="gain"/>
      </device_type_identifier>
      <device_info>
        <manufacturer>Cirrus Logic Inc.</manufacturer>
        <manufacturer_short>Cirrus Logic</manufacturer_short>
        <manufacturer_url>http://www.cirrus.com/</manufacturer_url>
        <model_short>Gain</model_short>
        <model_name>Gain</model_name>
        <model_number>Gain</model_number>
        <model_description>Gain</model_description>
        <model_url>http://www.cirrus.com/</model_url>
      </device_info>
    </device_type_info>
    <!--PROPERTY SCHEMA-->
    <device_property_schema>
    </device_property_schema>
    <!--CONFIGURABLE DATA SCHEMA-->
    <device_data_schema>
      <!--controls schema-->
      <controls>
        <control ruid="gain" type="level_db" default="0" label="gain">
          <restriction>
            <min value="-100"/>
            <max value="6"/>
          </restriction>
          <help>gain</help>
        </control>
        <control ruid="mute" type="mute" default="off" label="mute">
          <help>mute</help>
        </control>
      </controls>
      <!--ports schema-->
      <ports>
        <port ruid="output" domain="coyote_dsp_audio" type="source"
          label="channel output" />
        <port ruid="input" domain="coyote_dsp_audio" type="sink"
          label="channel input" />
        <port ruid="enable" domain="coyote_dsp_control_signal" type="sink"
          label="enable" />
      </ports>
    </device_data_schema>
  </device_type_interface>
</ningaloo_document>
```

4.3.2 <device_type_info>

The <device_type_identifier> element is exactly the same as found in the presentation XML file. In the <device_info> element, the important elements are the <model_name> and <model_short> which define the default name of the primitive when it is added to a design.

4.3.3 <device_property_schema>

This element is where design-time properties of the primitive are defined. The gain primitive example has no design-time properties. The following XML snippets are from other primitives.

```
<property type="bool" default="0" label="Default value" ruid="default_value" >
  <help>On/off switch</help>
</property>
<xpp:define name="channel_remap_restrictions">
  <restriction>
    <enum name="Left" value="0"/>
    <enum name="Right" value="2"/>
    <enum name="Center" value="1"/>
    <enum name="Left surround" value="3"/>
    <enum name="Right surround" value="4"/>
  </restriction>
  <help>Which audio channel should be mapped to this DAO channel?</help>
</xpp:define>
<property type="positive_int" default="0" label="DAO1 channel 0"
  ruid="DAO1_CHAN_0_REMAP">
  <xpp:insert name="channel_remap_restrictions" />
</property>
<property type="integer" default="8" label="Channel count" ruid="channel_count">
  <restriction>
    <enum name="8" value="8"/>
    <enum name="16" value="16"/>
  </restriction>
  <help>channel count</help>
</property>
```

The property type can be anything, although it must be correctly specified for the intended use of the property (usually used in the implementation XML file). For example, an “input_count” property that defines the number of input ports for a primitive must be of type “integer”.

The RUID value must be unique to the primitive.

The <help> will be available to designers using the *Device Properties* dialog. Typical property types are:

integer	Integer Value
positive_int	Positive Integer Value
float	Floating Point Value
bool	Boolean Value
string	Character String

The `<restriction>` element is optional for each property. Supported `<restriction>` types are:

<code><min value="n"/></code>	Minimum Numeric Value Allowed
<code><max value="n"/></code>	Maximum Numeric Value Allowed
<code><enum name="label" value="value"/></code>	One of the Possible Combo Box Choices

4.3.4 `<device_data_schema>`

This section defines the runtime behavior of the controls in the control panel, and some type information about the ports associated with the device.

The `<controls>` section contains zero or more `<control>` definitions, each of which defines the runtime characteristics of that control. The `<ruid>` specified for the control must match a `<ruid>` specified for a control in the control panel section of the presentation XML file. The restrictions that can be placed on a control's values are the same as supported for device properties as described above. The control type can be one of the following :

<code>string</code>	Control supports entry of a string value.
<code>float</code>	Control will produce a floating point value.
<code>integer</code>	Control will produce a signed integer value.
<code>enum</code>	Control will produce one of a set of enumerated values.
<code>unit32</code>	Control will produce an unsigned 32-bit value.
<code>unit64</code>	Control will produce an unsigned 64-bit value.
<code>bool, boolean</code>	Control will produce 0 or 1.
<code>switch off on, bypass, mute</code>	Control will produce 0 or 1.
<code>level, level db</code>	Control will produce a floating point value between -100 and +24.

The `<ports>` section contains zero or more `<port>` definitions, each of which defines salient characteristics of the port. The value, `ruid` must match the port RUID specified in the presentation XML file.

The domain attribute specifies what kind of wire can be connected to the port. The allowable choices are::

<code>coyote_dsp_audio</code>	A Single Channel of DSP Audio Data (typically as a 16-sample vector)
<code>coyote_dsp_control_signal</code>	A Single 32-bit Control Signal Variable

4.4 Implementation XML File (*.imp.xml*)

The implementation XML file defines all the information needed to create the object code and data structures required to produce and instances of the primitive in a design downloaded to the DSP. In general, this information includes:

- A set of object files (.O) containing the primitive code (Not all primitives have code, but most do)
- Pre-defined buffers for ports (if any)
- An expression that can be evaluated to determine the MIPS usage of the primitive
- Definitions of instance data structures. These are data structures that will be replicated for each instance of the primitive in a given design. At runtime, the primitive code will be passed pointers to these data structures by the microkernel.
- Script functions to translate runtime control values to the appropriate DSP variable values.

The following example on pages 18-20 shows an implementation XML file.

4.4.1 Example: gain.imp.xml

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2005 Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<ningaloo_document xmlns:xpp="www.peakaudio.com/xmlns/xmlpreprocessor"
  xmlns:xpr="www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator"
  xmlns:sxpr="www.peakaudio.com/xmlns/strudel/expressionevaluator">
  <device_type_implementation>
    <!--device type identifier-->
    <device_type_identifier>
      <family value="coyote_dsp"/>
      <type value="gain"/>
    </device_type_identifier>

    <!-- predefined module information -->
    <MCTsymbol value="X_P_BY_Gain_PCT" />
    <MIPS value="((sample_rate/block_size)*(20+(2*block_size)))/10000000.0"/>
    <objectFiles>
      <file value="./gain.0"/>
    </objectFiles>

    <!-- default control port buffer assignments -->
    <ports>
      <port ruid="enable" buffer="I_VY_one" />
    </ports>

    <!-- data structures -->
    <structures>
      <!-- public instance data -->
      <structure>
        <name>public_data</name>
        <member>
          <name>public_y_data</name>
          <type>public_y_data_t</type>
          <memory>Y</memory>
        </member>
      </structure>
      <structure>
        <name>public_y_data_t</name>
        <member>
          <name>gain</name>
          <type>fract(8.24)</type>
          <!-- the following element indicates that this data member is
              associated with an external controllable variable called "_gain"
          <control_var>_gain</control_var>
          <access>write</access> <!-- read, write, or both -->
          <initialize>1.0</initialize>
        </member>
      </structure>
      <!-- private instance data -->
      <structure>
        <name>private_data</name>
        <member>
```

```

        <name>private_y_data</name>
        <type>private_y_data_t</type>
        <memory>Y</memory>
    </member>
</structure>
<structure>
    <name>private_y_data_t</name>
    <member>
        <name>enable_ptr</name>
        <type>bool*</type>
        <port>enable</port>
    </member>
    <member>
        <name>input_ptr</name>
        <type>integer**</type>
        <comment>input buffer ptr</comment>
        <port_stride>input</port_stride>
    </member>
    <member>
        <name>input_stride</name>
        <type>integer</type>
        <comment>stride</comment>
        <initialize>DESIGN_BRICK_MODULO</initialize>
    </member>
    <member>
        <name>output_ptr</name>
        <type>integer**</type>
        <comment>output buffer ptr</comment>
        <port>output</port>
    </member>
    <member>
        <name>output_stride</name>
        <type>integer</type>
        <comment>stride</comment>
        <port_stride>output</port_stride>
    </member>
    <member>
        <name>block_size</name>
        <type>integer</type>
        <comment>block size</comment>
        <initialize>DESIGN_BRICK_SIZE</initialize>
    </member>
    <member>
        <name>private_gain</name>
        <type>integer</type>
        <comment>gain factor</comment>
    </member>
</structure>
</structures>

<!-- include python scripts for translating external GUI control values to
    DSP control word values. NOTE: since the CDATA contents will be copied
    directly into a python script, and python is whitespace-sensitive, it
    is IMPERATIVE that the lines of python script between the CDATA begin/end
    start out NOT indented.

```

```
-->
<poke_crunch_function>
  <![CDATA[
# python function for crunching input control values
if( mute ) :
  _gain = -100
else :
  _gain = gain
  ]]>
</poke_crunch_function>

</device_type_implementation>
</ningaloo_document>
```

4.4.2 <device_type_identifier>

This element must exactly match the information in the same element in the presentation and interface XML files.

4.4.3 Module information

This section contains several elements:

<MCTsymbol>

This is the publicly defined address of the module call table for the primitive code. The format of the module call table is described in the Data Structures section below. The DSP Conductor generates an ".extern" statement for this symbol.

<MIPS>

An expression that can be evaluated to determine how much DSP computing resource this primitive consumes. Computing resources are measured in million instructions per second (MIPS). The expression should be as accurate as possible. Inaccuracy should err on the conservative side (over-estimating MIPS requirements). The expression can make use of variables such as:

block_size - The size of a block of samples (a project design-time property, typically 16)

sample_rate - The maximum sample rate this project is expected to handle

input_count - Number of input audio ports to a primitive with variable number of input ports.

output_count - Number of output audio ports to a primitive with variable number of output ports.

<objectFiles>

A list of <file> elements enumerating the object files (.O) included when instances of this primitive appear in a design. Each object file can be specified with absolute path or a path relative to the location of this *imp.xml* file.

Default Buffer Assignments

For some primitives it makes sense to have some or all of the ports assigned to a default buffer to handle the case where the designer does not connect a wire to one or more ports. This is useful for control signal ports, where a default value of *I_VY_one* or *I_VY_zero* indicates that the port is to be wired to a 1 or a 0 constant if it is not specifically connected in a design.

Except for a small number of special-purpose, system-defined primitives, audio data ports are rarely given default buffer assignments.

Data structures

Data structures are defined in the implementation XML file using a particular set of XML elements that comprise a structure definition language (*strudel*). The elements of that language are given in the table below:

<code><structure></code>	Begins a structure (type) definition. Analogous to a C struct declaration.
<code><member></code>	Defines a member of the enclosing structure. Analogous to a member variable of a C struct.
<code><name></code>	Specifies the name of the enclosing structure or member.
<code><type></code>	<p>Specifies the data type of the enclosing member. Allowed values are:</p> <ul style="list-style-type: none"> - integer - float - bool - <i>fract(integer_part.fractional_part)</i> - A signed fixed point value with the specified number of bits to represent the <i>integer_part</i> and the <i>fractional_part</i>. Can take either an integer or a floating point initial value. - <i>ufract(integer_part.fractional_part)</i> - An unsigned fixed point value with the specified number of bits to represent the <i>integer_part</i> and the <i>fractional_part</i>. Can take either an integer or a floating point initial value. - buffer - A member declared as buffer indicates that it is supposed to be a pointer to a temporary buffer. - void - A member declared as void will not generate any output by the compiler. It must contain a set of <i><item></i> members whose values will be inserted verbatim by the compiler into the assembly language source file for the project. - User-defined type (some other structure name) - An instance of that structure will be inserted. - A pointer to any of the above (denoted by <i>type*</i>).
<code><control_var></code>	Indicates the RUID of a runtime control that controls this data member. The control RUIDs are defined in the interface XML file.
<code><access></code>	For members with the <code><control_var></code> attribute, this specifies whether the control is a read, write, or both control. The default (if no <code><access></code> element is specified) is write.
<code><initialize></code>	An initial value for the data member. The value must be a constant.
<code><sxpr:initialize></code>	An initial value for the data member. The value is an algebraic expression that can contain references to any of the device or project properties.

<code><memory></code>	Indicates whether the member is to be allocated in X, Y, or XY memory.
<code><port></code>	Indicates the RUID of a port defined for the device. Indicates that the enclosing data member is to be initialized by a pointer to a pointer (integer**) to the buffer assigned to that port .
<code><port_stride></code>	Indicates the RUID of a port defined for the device. Indicates that the enclosing data member is to be initialized to the predefined stride value appropriate for the buffer assigned to that port
<code><comment></code>	A comment - currently not used.
<code><memory_pool></code>	For a member of type buffer, this indicates whether the buffer is to be allocated in internal, external, or heap memory. Heap memory is internal memory that is not statically allocated when a project is compiled, but dynamically allocated at runtime. In CobraNet applications only internal memory is available. The value of <code><memory_pool></code> can be a list of the possible values, joined by the ' ' character, to indicate that the buffer may be allocated in whichever pool has enough memory available. First the compiler attempts to allocate internal, then heap, and finally external memory.
<code><align></code>	For members of type buffer this optional member specifies any allocation alignment requirement. For example, if a buffer must be on a modulo 256 address, set this element to 256.
<code><size></code>	For members of type buffer this indicates the size (integer constant) of the buffer to be allocated.
<code><sxpr:size></code>	Like <code><size></code> but this allows an expression to be used instead of an integer constant. The expression can reference any design or device properties.
<code><dimension></code>	Indicates that the enclosing member is an array, and has the specified number of elements.
<code><sxpr:dimension></code>	Like <code><dimension></code> but allows an expression to determine the number of elements.
<code><label></code>	Gives the name of a label to be placed in the emitted source file before emitting the data for the enclosing member or structure. An optional attribute of <code>public="1"</code> can be specified if the label needs to be made public for some reason.
<code><sxpr:label></code>	An expression that evaluates to a string, to be used as a label as described above.
<code><buffer_memory></code>	Same as <code><memory></code> . Used to indicate where members of <code><type></code> buffer are allocated.

For a primitive, the presence of one of the following structure names indicates that that type of data is present for the primitive:

public_data - Represents the public read/write data of the primitive. Any primitive that has write-access runtime controls must have a *public_data structure*. This structure, if defined, must be in Y memory.

public_RO_data - Represents the public read-only data of the primitive. This structure, if defined, must be in Y memory.

private_data - Represents any private data required by an instance of this primitive.

Some of the elements allow expressions to be used for values. Expressions can consist of an algebraic expression with constants, variables, and operators. String constants should be enclosed in single quotes. Numeric constants are just numbers. Although, as per XML syntax, the entire expression must be enclosed in double quotes, variable names themselves are not enclosed in quotes.

The available variable values are the RUIDs of all device properties, the RUIDs of all design properties, and a couple of special RUIDs:

<code>pfx</code>	string	A prefix that the compiler generates that is unique for the specific instance of an algorithm. This variable can be used to generate unique strings for an instance a primitive.
<code>sample_rate</code>	float	The Maximum Sample Rate (project property)
<code>block_size</code>	integer	Number of Samples in a Brick (project property)

The operators that are allowed are the standard arithmetic operators +, -, *, /, and %.

Parentheses may be used to indicate precedence.

A '+' operator between string operands indicates concatenation.

Runtime Control Scripts

The implementation XML file can define python scripts used to translate runtime control values into 32-bit (coefficient) values appropriate for use by the DSP code. The translation can be as simple as just a comment (no code), in which case the control value is sent directly to the DSP.

Within the python script environment, DSP Conductor assigns variables corresponding to control RUIDs defined in the interface XML file. In the *gain.imp.xml* example, the variable "gain" is the value of the runtime control (see the RUID assigned to the *level_db knob* in *gain.if.xml*). In addition, DSP Conductor creates a map of all design and device properties for access by the script.

Variables need to be created for any `<control_var>` names in any read-write structure definitions in the implementation XML file. For example, in *gain.imp.xml*, "g" and "ramp" are DSP control variables or coefficients in the *public_data structure*.

If a design or device property is needed by the script, it should be accessed as follows:

```
foo = float( property[ " foo " ] )
```

For example, in *gain.imp.xml*, the "sample_rate" and "block_size" design properties are needed in order to compute the block rate, which in turn is used to compute the gain ramp DSP coefficient. The "scale_factor" design property supports runtime sample rate changes.

As shown in *gain.imp.xml*, standard python math functions can be used to compute 32-bit DSP (coefficient) values from control values.

Variables also need to be created corresponding to read-only control RUIDs defined in the interface XML file. For example, in *gain.imp.xml*, the variable "clip" is the value of the runtime control (see the RUID assigned to the clip indicator in *gain.if.xml*).

Note that, in *gain.imp.xml*, since both 32-bit DSP variable and runtime control are named "clip", there is no need to explicitly create a "clip" runtime control variable. The 32-bit DSP value is implicitly copied from the DSP to the control.

For an example of how runtime control variables are explicitly created and assigned values that are computed from 32-bit DSP values, see the *meter_true_peak_and_RMS.imp.xml* file located in ...*\program files\cirrus logic\dspconductor\plugins\coyote\devices* directory.

For debugging purposes, script variables can be printed to DSP Conductor's *Debug* pane as follows:

```
message.string_set( "foo = " + str( foo ) )
```

NOTE: Since the *CDATA* contents will be copied directly into a python script, and python is whitespace-sensitive, it is imperative that the lines of python script between the *CDATA* begin/end start out not indented.

<poke_crunch_function> - the *<![CDATA[* element defines a python script that will be executed whenever any of the runtime controls are gestured.

<peek_crunch_function> - the *<![CDATA[* element defines a python script that will be executed on a polling cycle to retrieve values from the DSP for display by read controls at runtime. The polling frequency is a design property.

5. Primitive Code

Algorithm code runs in the environment provided by the DSP Conductor microkernel. Each algorithm has 3 entry points:

initialization - entry point that will be called once when the microkernel is restarted

block - called whenever a block-sized amount of data is available to be processed. All algorithms in a design are called to process the same block of data before the next block of data is processed.

background - called by the microkernel whenever there is nothing else to do. This entry point can monitor public variable changes or do other background computations

The algorithm can have both *public* and *private* data. Public data can be accessed by a host microcontroller or other process external to the DSP. Private data is used only by the algorithm code running on the DSP. Public data is further divided into read/write and read-only regions.

The algorithm must be written such that its code and data can be properly linked to run with the DSP Conductor microkernel. For code written in assembly language, this means correct use of segment definition macros, as described later. Algorithm code is usually contained in a single source (.a) file and assembled into a single object (.O) file. The object filenames are referenced in the implementation XML file, as described later.

Algorithms as used in DSP Conductor are much like C++ class definitions. Each time a primitive device is added to the design canvas, it implies a new instance of the class. The actual code of the algorithm is only included in the final product one time, just as class methods are only implemented once, but are available for use by all instances of the class. Similarly, any data that is defined in an algorithm source file is only included once, and is therefore analogous to C++ class static data members. For DSP Conductor primitives, instance-specific data must be defined in the implementation XML file described below. DSP Conductor creates a new set of such data for each instance of a particular algorithm.

Coding Conventions:

- Any data defined in the algorithm source files is only included once in the final module and is shared by all instances of the algorithm. All such data declarations must use the “_cc” suffix (e.g., .ydata_cc) so that it can be properly linked with microkernel files.
- Algorithm code should be in a .code_cc segment.
- An algorithm must define, in Y memory, a 3-word table specifying the entry points into the code. The beginning of the table must be defined by a public symbol (it will be referenced by other code created by DSP Conductor).
- Each entry point to the algorithm can expect *i6* to point to the instance public read/write data, *i7* to point to the instance private data, and *i5* to point to the public read-only data. These index registers may be NULL (0) if the algorithm doesn't have public or private instance data.
- Audio data buffer pointers in the instance data are initialized with indirect addresses. That is, an instance data pointer to an audio buffer is actually a pointer to a pointer to the real data. Audio buffers themselves are always in Y memory, and are allocated on block-sized modulo boundaries.
- Audio data buffers may or may not be interleaved. Code that steps through an audio buffer should use the *nm* registers of the DSP to increment through the buffer, and the *nm* register should be loaded from a data member that is initialized using the <port_stride> element in the implementation XML.

- The number of samples contained in a single-channel I/O buffer (block size) is typically 16, but can be changed via a DSP Conductor project property. Therefore, algorithm code should not use fixed constant-size assumptions about samples per buffer, but should have an instance variable (either public or private) that contains the number of samples per I/O buffer. The discussion of the Implementation (*.imp.xml*) file shows an example of this.
- Most DSP Conductor primitives are written so that they can run in either the AVR or CobraNet environments. To enable this, algorithm code must not use registers *i8-i11* or *nm8-nm11* as these are reserved for OS and microkernel use.

Sample code:

The following is the code for the gain primitive algorithm, in its entirety.

```
#####
#
# Gain_Primitive
#
# This is the source for the object file that gets linked into
# the project when the primitive is instantiated
#
#####

#####
#
# public data (MCV) in YMEM
#
# .dw (0x00000000)          # log gain (signed 8.24, range -inf:+6)
#                          (anything less than ~-90dB (0xa600) == -infdB)
#
# private data in YMEM
#
# .dw (enable_ptr)          # enable/disable YMEM*
# .dw (X_VY_IOBuffer_0_Ptr) # Input YMEM**
# .dw (Mod<block_size>)     # Input Modulo
# .dw (X_VY_IOBuffer_2_Ptr) # Output YMEM**
# .dw (Mod<block_size>)     # Output Modulo
# .dw (<block_size>)        # block size
# .bss 1                    # internal linear gain value
#
#####

#####
#
# Common Data
#
# Includes subroutine table as well as any other variables
# that are shared by all instances of this primitive
# (a la C++ static class members)
#
#####

.public X_P_BY_Gain_PCT

.ydata_cc
X_P_BY_Gain_PCT          .dw I_P_S_Gain_Init
                        .dw I_P_S_Gain_Block
                        .dw I_P_S_Gain_Bkgnd

I_P_log10_to_log2        .dw .f2b(.log2(10)/(20*2))

#####
# Common Code
#####

.code_cc

#####
```

```

I_P_S_Gain_Init
    i7 = i7 + (6)
    a0=0
                                ymem[i7] = a0
    ret

#####

I_P_S_Gain_Block
    mr_sr = (0)

# get public (i6) and private (i7) data
    i0 = ymem[i7]; i7+=1                # check enable/disable
    i5 = ymem[i7]; i7+=1                # i5: Input
    a0 = ymem[i0]
    a0&a0
    if (a==0) jmp >end

    nm5 = ymem[i7]; i7+=1
    i0 = ymem[i7]; i7+=1                # i0: Output
    nm0 = ymem[i7]; i7+=1
    i4 = ymem[i7]; i7+=1                # i4: Blocksize
    y0 = ymem[i7]; i7+=1

# indirect I/O pointers from OS
    i5 = ymem[i5]
    i0 = ymem[i0]

# Do the Dew...
                                x0 = ymem[i5]; i5+=1
    a0 = x0*(unsigned)y0

    do (i4), >
                                x0 = ymem[i5]; i5+=1
%:  a0 = x0*(unsigned)y0; ymem[i0] = a0; i0+=1

%end
    ret

#####

I_P_S_Gain_Bkgnd
# translate public log gain to private linear gain
                                x0 = ymem[i6]                # y0: log gain
                                i7 = i7 + (6)
    call I_S_20log10_to_Linear
                                ymem[i7] = a0h                # a0h: linear gain

    ret

#####
# input:  x0 (max: +6dB (0x06000000))
#         (anything less than --90dB (0xa600) == -infdB)
# output: a0
#####

I_S_20log10_to_Linear
# convert from 20*log10 to log2
# and move from 8.24 to 9.23 (extra shift in log10_to_log2 constant)
    a0 = (0x0f80)                # bias == 32-1
    y0 = ymem[I_P_log10_to_log2]
    a0 += x0*y0
                                x0 = a0

# convert back to linear (2^x)
    logexp X=exp(x0)  Y=sm(x0)
    nop
    x0,y0 = logexp
    a0 = -x0*y0
    a0 = a0 << 1

    ret

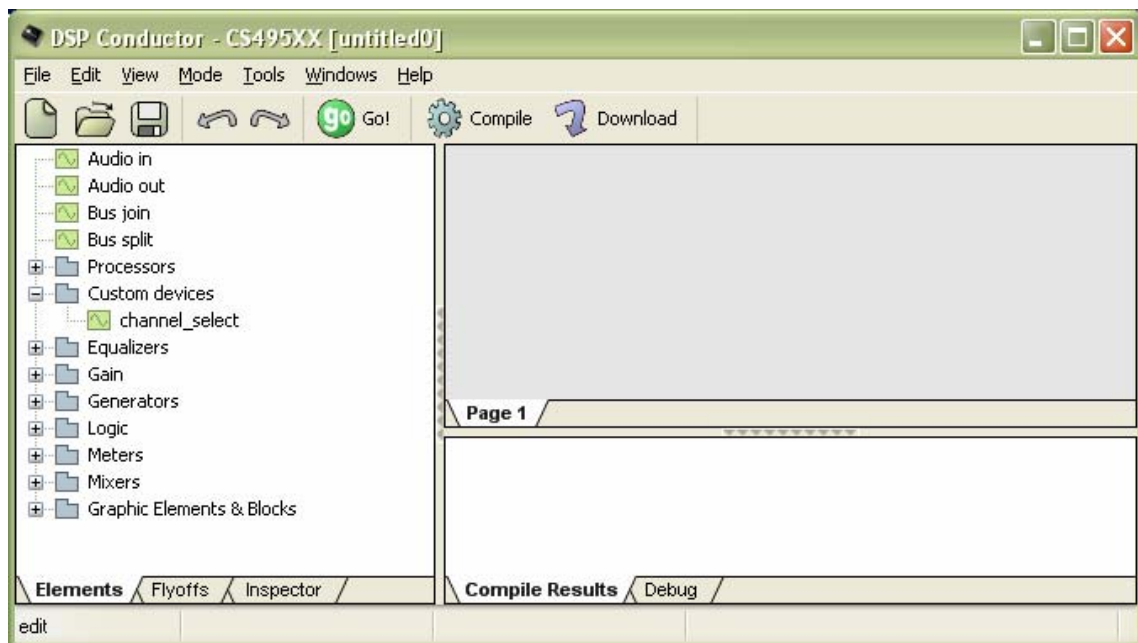
```

6. *channel_selector*, a DSP Conductor Primitive Tutorial

In this section we will walk, step by step, through the process of creating a new DSP Conductor primitive. We call the primitive "*channel_selector*". This primitive selects among several input audio channels to generate a single output channel. The active input channel is selected by a runtime control.

This procedure assumes that you have installed DSP Conductor in the C:\CS495XX\ directory.

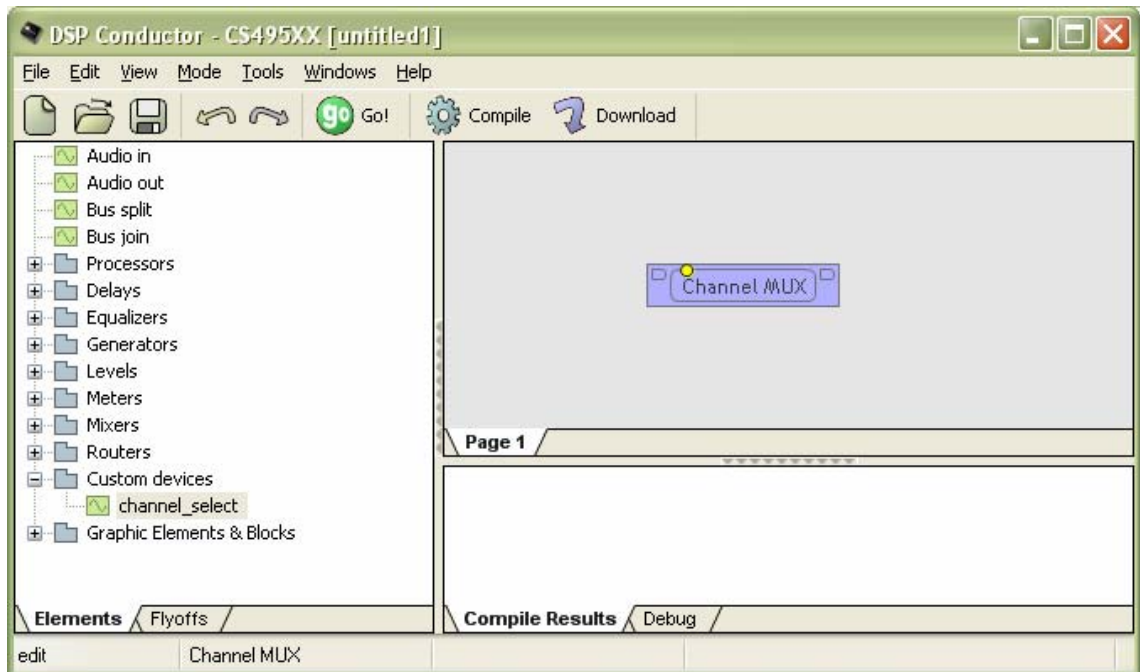
- Step 1. Make sure you have created a "custom" directory under the c:\cs495xx\DSPConductor\plugins\coyote\devices directory.
- Step 2. Start with something similar. In this case, copy all of the *gain*.xml* files from the c:\cs495xx\ DSPConductor \plugins\coyote\devices directory to the c:\cs495xx\ DSPConductor \plugins\coyote\devices\custom directory.
- Step 3. Rename all the c:\cs495xx\CoyoteCAD\plugins\coyote\devices\custom\gain.*.xml files to c:\cs495xx\ DSPConductor \plugins\coyote\devices\custom\channel_select.*.xml.
- Step 4. Start DSP Conductor. In the device palette you should see a folder called *Custom devices*. In that folder you should see a device called *channel_select*. If you don't see that, check your work from the preceding steps. Try to drag a *channel_select* onto the canvas. You will not be able to. DSP Conductor will give you hints about why you can not. Click on the *Debug* tab. You will see a message saying that the custom device menu was searched. You'll see an exception message that reads something like "exception creating device coyote_dsp channel_select : no device interface".



Looking for the interface definition, DSP Conductor opened *channel_select.if.xml* but could not find the interface specification for the device. But wait! you say. We just created one! Well, yes and no. We created a file with the correct name, but it is a copy of the gain primitive file. Internally that file still claims that it is the interface of a primitive called *gain*. We'll fix that in the next step. For now, close DSP Conductor. Whenever you need to make changes to either the interface or presentation XML files for a primitive, you'll need to restart DSP Conductor to make it aware of the changes.

When we get to the point of changing the implementation XML file we no longer have to do that, as DSP Conductor rereads the implementation XML file whenever the *Compile* function is invoked.

- Step 5. Open the new *channel_select.if.xml* file in the editor of your choice. For editing XML files, the Microsoft Visual Studio editor is an excellent choice, but you may use any editor you like. Find the `<device_type_identifier>` element, and change the `<type>` subelement's value attribute from "gain" to "channel_select". While you're in here, we'll go ahead and change the values in the `<device_info>` section. For example, change the `<model_short>` and `<model_name>` from *Gain* to *Channel MUX*. Save the changes. Reopen DSP Conductor and try to drag a *channel_select* device onto the canvas. Again it fails! Click on the *Log* tab and look at the message. Same message, except now it says "no presentation"! What gives?
- Step 6. There are three XML files, and they all have to have the appropriate `<device_type_identifier>` element. Shut down DSP Conductor, and open the *channel_select.pres.xml* and *channel_select.imp.xml* files in your editor, and make the `<type>` in the `<device_type_identifier>` be *channel_select* for both of them. Save the changes. Restart DSP Conductor and try to drag a *channel_select* device onto the canvas. It should succeed in creating a device this time, although it doesn't yet look the way we want.



Shut down DSP Conductor so we can go on to make the next changes.

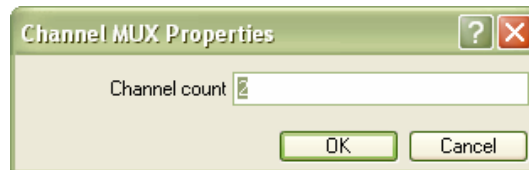
- Step 7. As we change the way our *channel_select* primitive looks in DSP Conductor, we will be changing both the interface (*.if.xml*) and the presentation (*.pres.xml*) files in tandem - they both have to agree on many things in order for the primitive to work properly in DSP Conductor. Therefore, you may want to keep both files open in your editor for now.

The first thing we want to do is allow the *channel_select* primitive to have more than one audio input port. In fact, we'd like to make the number of ports be a design-time property of the device, with a default of *two*. For this step we need to do two things: add a device property for the input channel count, and then use that property in other parts of the definition to affect the number of input ports.

First let's add the property. In the interface XML file there is an empty element called `<device_property_schema>`. We need to add an element for our new property. We also want the property to be an integer in the range [1,16]. To do this we need to add the following XML elements inside the `<device_property_schema>`:

```
<property type="integer" default="2" label="Channel count" ruid="input_count">
  <restriction>
    <min value="1"/>
    <max value="16"/>
  </restriction>
  <help>Input channel count</help>
</property>
```

- Step 8. Save the interface XML file and restart DSP Conductor. Drag a *channel_select* device onto the canvas, right-click on it, and choose *Device properties*. You should see the following dialog:



- Step 9. Now we need to use the newly defined property to change the number of input ports the device shows on the screen. To change the way a primitive looks on the screen, changes must be made to the presentation XML file. Go down to the where the comment says "device layout" and find the element that says

```
<xpp:insert name="coyote_dsp_1_by_1_presentation"/>
```

This element defines the device as having 1 input and 1 output port. We want to change it to have a variable number of inputs and one output. DSP Conductor already has a definition for this type of device. To invoke this, make the following change

```
<xpp:insert name="coyote_dsp_x_by_1_presentation"/>
```

and save the file. Restart DSP Conductor, drag the *channel_select* onto the canvas, and you should see that it has 2 input ports on the left side and one output port on the right side. Right-click on the device, choose *Device properties*, and change the channel count to 3. You should now see 3 input ports. Wasn't that easy?

- Step 10. In DSP Conductor, different classes of primitives have different cosmetic appearances. Different primitives that belong to the same class share the same cosmetic appearance. This is accomplished by the element in the presentation XML file described in the *Device args* section above. Our *channel_select* primitive started from the *gain* presentation XML, but a gain primitive is part of the "level" class of primitives, while our new primitive really should be in the "router" class. To make this simple change, edit the presentation XML line that reads

```
<xpp:insert name='device_args_level' />
```


and change it to read:

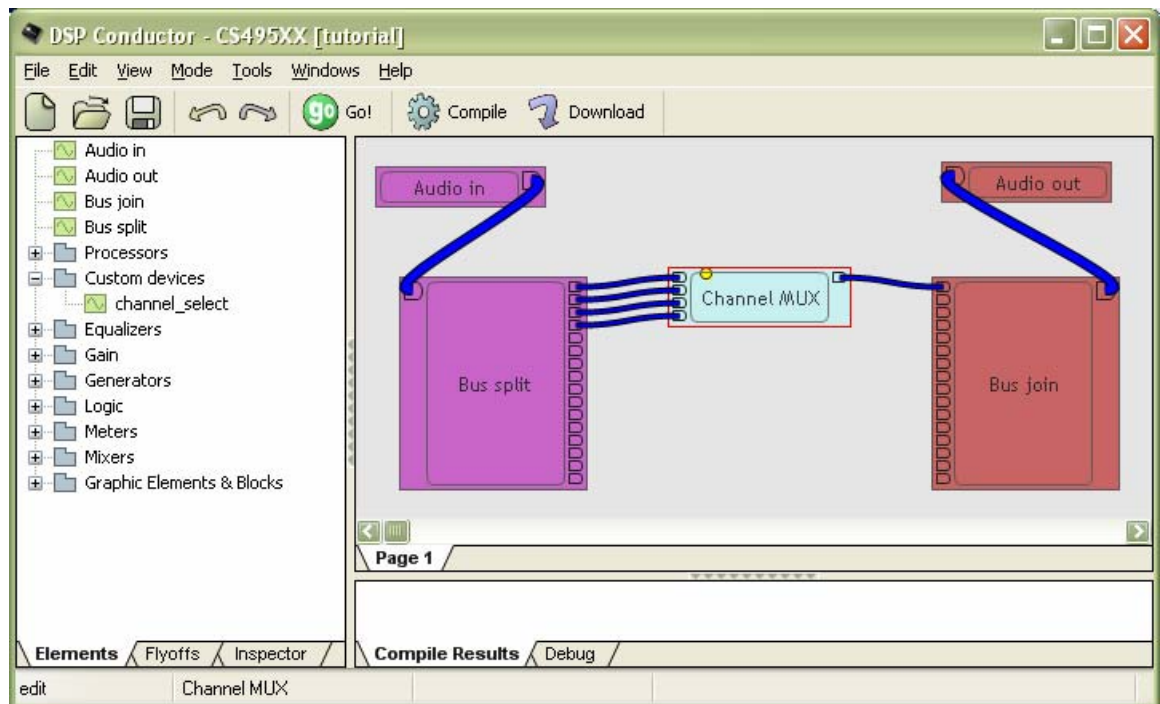
```
<xpp:insert name='device_args_router' />
```

Restart DSP Conductor and create a *channel_select* device to see the effect of this change.

Step 11. Next we need to change the interface XML file so that the port definitions match the variable number of ports supported by the presentation XML file. Shut down DSP Conductor and open the interface XML file. Find the `<ports>` element. Since we copied this file from the gain primitive, there should be three port definitions: *output*, *input*, and *enable*. At this time the output and enable port definitions are fine, but we've changed the presentation to support a variable number of input ports, so we need to make a complementary change here. To do that, remove the `<port>` definition for the input port (the one with *ruid*="input"). We need to replace that with some sort of expression based on the device property "input_count" we defined in Step 7. To do this we'll use the `<xpp:>` and `xpr:>` XML pre-processor constructs. Enter the following lines after the output `<port>` definition:

```
<xpp:repeat index="i" first="1" last="input_count">
  <port xpr:ruid="'input_' + i" domain="coyote_dsp_audio"
    type="sink" xpr:label="'audio channel(' + i + ')"
    xpr:index="i-1" />
</xpp:repeat>
```

This code repeats a `<port>` definition for "input_count" number of times, assigning a computed RUID and label for each port as it goes. To test this, start DSP Conductor, add our new device to a design, add other devices to the design, and check that you can connect all of the ports of the new *channel_select* device to ports on other devices. Change the device property for channel count and connect the new channels too. You should be able to make up a design something like the following:



Step 12. Time to start writing the actual code for our primitive now. This primitive's function is simple; the code required will be small. The design needs to support a data structure that contains an output buffer pointer and n input buffer pointers where n is the number of input channels. It should also have a variable that is publicly available for runtime control of the input channel selection. See Appendix A for source that correctly implements this design. Follow the instructions in the appendix to assemble the source and create the object file needed for building a DSP Conductor project with the primitive.

Step 13. Next we need to change the control panel for the `channel_select` device. The gain XML files we started with will not produce an appropriate control panel. Double-click on the *Channel MUX* device in the design canvas to see the gain control panel. Then close DSP Conductor so we can change the controls.

Step 14. The visible parts of the control panel are defined in the presentation XML file. In that file find the section that starts with

```
<xpp:define name="control_panel">
```

Notice that there is a single `<column>` containing two controls, a `"ctl_button_wide"` (for the mute) and a `"ctl_knob_with_textbox_and_label"` for the gain control. We want to change this to a single control that allows the user to choose which of the input channels is to be mux'ed to the output channel. First, remove the two lines dealing with the mute control (the first two lines after the `<column>` element). Then change the remaining lines to look like the following:

```
<xpp:define name="ctl_args" >
  <member name='text' value='Select' />
  <member name='hover_text' value='Select channel' />
  <member name='ruid' value='select' match='true'/>
  <member name='color_rgb' value='255 200 180' rule='color_rgb' once='true' />
  <xpp:evaluate expression="ctl_options=''" />
  <xpp:repeat index="c" first="1" last="input_count" >
    <xpp:evaluate expression="ctl_options=ctl_options+c+':'" />
  </xpp:repeat>
  <member name='ctl_options' xpr:value='ctl_options' />
</xpp:define>
<xpp:insert name="ctl_combobox_with_label" />
```

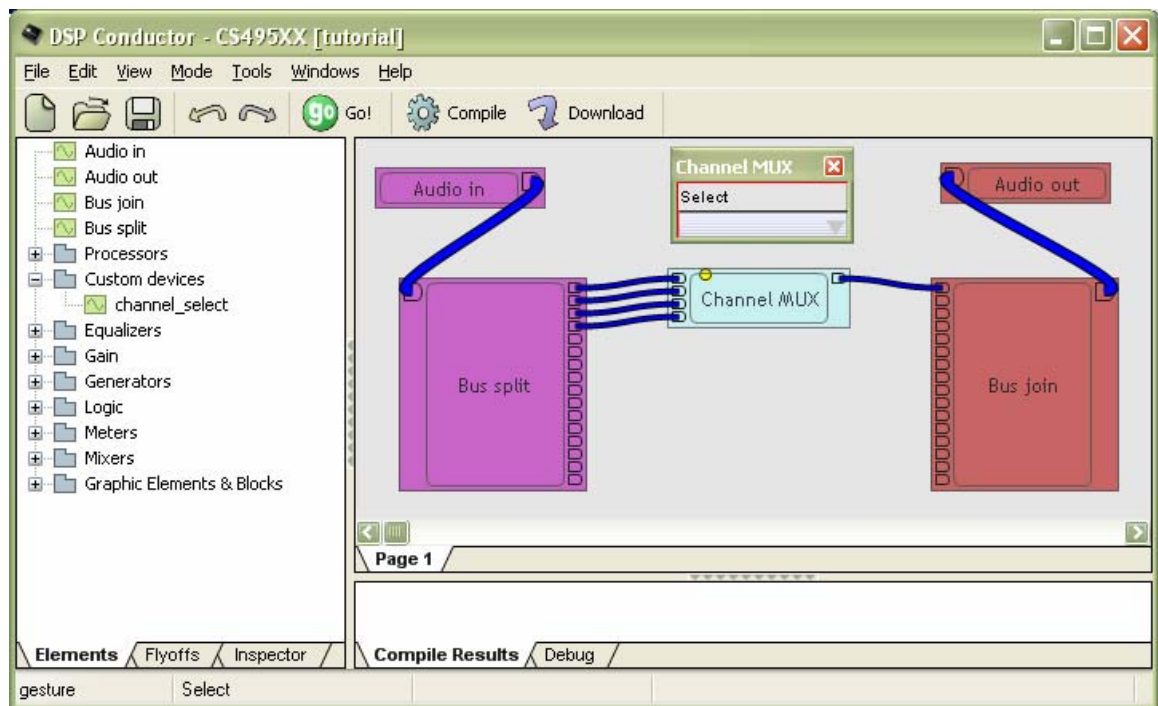
The last line changes the control from a knob to a combobox control. The lines before that define parameters that are then included in the combobox control definition. The `xpp:` code that creates a value for the `"ctl_options"` member name is creating a string in the form `"1:2:3:...:n:"` - this string is the list of allowable values for the combobox control to take at runtime. The `<xpp:repeat>` expression creates a string that has choices from 1 to the number of input channels specified by the device property. Save these changes.

Step 15. Another part of the control panel definition is in the interface XML file, and this part must complement the definition in the *presentation* XML file. Edit the *interface* XML file and find the section that contains the `<controls>` element. Again, remove the existing `<control/>` specifications for the con-

trols left over from the original *gain* interface file. Then add the following lines:

```
<control ruid="select" type="enum" default="1" label="Channel">
  <restriction>
    <xpp:repeat index="i" first="1" last="input_count" >
      <enum xpr:name="'Channel '+i" xpr:value="i" />
    </xpp:repeat>
  </restriction>
  <help>Selected input channel</help>
</control>
```

This definition makes an enum control named "select". The `<restriction>` clause uses an `<xpp:repeat>` construct to create separate restrictions for the different possible values. Save these changes. Start DSP Conductor and create a *channel_select* device. Double-click on it. You should see a control panel that looks like:



Step 16. Almost done now. We still have to edit the implementation XML file (which we copied from *gain*) to emit the data structures expected by the primitive algorithm code we created in Step 12. Open *channel_select.imp.xml* in your editor. Change the `<MCTsymbol>` to match the symbol name used in the assembly code:

```
<MCTsymbol value="X_P_BY_channel_select_PCT" />
```

While we're here, go ahead and change `<MIPS>` to use the formula we drive from analysis of the assembly code:

```
<MIPS value="( (sample_rate/block_size) * (19+ (2*block_size))) /1000000.0"/>
```

And of course we need to change the name of the object file:

```
<objectFiles>
    <file value="./channel_select.O"/>
</objectFiles>
```

Note that the object file pathname is a relative path, and that it uses forward slash (`/`) instead of backslash - this path is used in a makefile which uses forward slash-delimited pathnames.

Next we need to adjust the XML that emits data structures. Since our primitive has one runtime-controllable variable, it needs to have one read/write public variable. The other data structures used to represent input and output buffer addresses and channel counts can be private variables. Start by deleting all the lines between the `<controls>` and `</controls>` elements. Add the following lines instead.

```
<structure>
    <name>public_data</name>
    <member>
        <name>select</name>
        <type>integer</type>
        <control_var>select</control_var>
        <access>write</access>
        <initialize>1</initialize>
    </member>
</structure>
```

This defines the public variable that the algorithm uses to determine which of the possible input channels to copy to the output. The `<control_var>` section tells DSP Conductor to connect the runtime control called "select" to this variable. The fact that the member `<name>` field is also "select" is irrelevant.

Next add the following structure defining private data. This is a structure that contains a structure; it tells DSP Conductor that the contained structure is to be allocated in Y memory. Private variables are free to reside in X, Y, or XY memory. Since we have written *channel_select.a* to expect all variables in Y memory, that's where we need DSP Conductor to put them:

```
<structure>
    <name>private_data</name>
    <member>
        <name>private_y_data</name>
        <type>private_y_data_t</type>
        <memory>Y</memory>
    </member>
</structure>
```

Now add the data structure definition for the private variables. The layout, type, and order of this definition must match the expectation from the algorithm code:

```
<structure>
  <name>private_y_data_t</name>
  <member>
    <name>enable_ptr</name>
    <type>bool*</type>
    <port>enable</port>
  </member>
  <member>
    <name>max_channel</name>
    <type>integer</type>
    <sxpr:initialize>input_count</sxpr:initialize>
    <comment>input channel count</comment>
  </member>
  <member>
    <name>output_ptr</name>
    <type>integer**</type>
    <comment>output buffer ptr</comment>
    <port>output</port>
  </member>
  <member>
    <name>output_stride</name>
    <type>integer</type>
    <comment>output buffer stride</comment>
    <port_stride>output</port_stride>
  </member>
  <member>
    <name>input_ptrs</name>
    <type>input_ptr_t</type>
    <dimension>input_count</dimension>
  </member>
</structure>
<structure>
  <name>input_ptr_t</name>
  <member>
    <sxpr:name>'input_ptr'+(i+1)</sxpr:name>
    <type>integer**</type>
    <comment>input buffer ptr</comment>
    <sxpr:port>'input_'+(i+1)</sxpr:port>
  </member>
  <member>
    <sxpr:name>'input_stride'+(i+1)</sxpr:name>
    <type>integer</type>
    <comment>input buffer stride</comment>
    <sxpr:port_stride>'input_'+(i+1)</sxpr:port_stride>
  </member>
</structure>
```

This structure has:

- A variable that maps to the control signal port *"enable"*.
- An integer for the max input channel number (used as a sanity check in the algorithm code).
- A pointer to the output buffer pointer, linked to the output port (*<port>output</port>*).

- d) A stride value for the output buffer which will be initialized by DSP Conductor appropriately depending on how the device is wired in a design
- e) An array of input buffer pointers, each of which is a pointer to a buffer pointer and a stride value. The array is of dimension "*input_count*". The *i* in the port expression is an integer value created by iterating from 0 to *dimension-1*. Since port names defined in the interface file start with 1, the (*i*+1) sub-expression is necessary.

Step 17. All the parts should be in place now. Create a new design that uses the *channel_select* device and click on the *Compile* and then the *Deploy* button. Once the application is deployed you should be able to click on the *Connect* button and try the control panel out, switching input channels.

Appendix A. *channel_select.a* Example

Assembly source for the *channel_select* primitive. To use it, copy these lines into a file called *c:\cs495xx\CoyoteCAD\plugins\devices\custom\channel_select.a*. To assemble the file, start a console window from the SDK menu, change the current directory to the directory containing the source, and enter

casm channel_select.a -L -H

on the command line. That should create a *channel_select.O* file in the same directory, which is where the tutorial expects to find it.

```
#####
#
# Gain_Primitive
#
# This is the source for the object file that gets linked into
# the project when the primitive is instantiated
#
#####

#####
#
# public data (MCV) in YMEM
#
# .dw (0x00000000)          # log gain (signed 8.24, range -inf:+6)
#                          (anything less than ~-90dB (0xa600) == -infdB)
#
# private data in YMEM
#
# .dw (enable_ptr)         # enable/disable YMEM*
# .dw (X_VY_IOBuffer_0_Ptr) # Input  YMEM**
# .dw (Mod<block_size>)    # Input  Modulo
# .dw (X_VY_IOBuffer_2_Ptr) # Output YMEM**
# .dw (Mod<block_size>)    # Output Modulo
# .dw (<block_size>)       # block size
# .bss 1                   # internal linear gain value
#
#####

#####
#
# Common Data
#
# Includes subroutine table as well as any other variables
# that are shared by all instances of this primitive
# (a la C++ static class members)
#
#####

.public X_P_BY_Gain_PCT

.ydata_cc
X_P_BY_Gain_PCT          .dw  I_P_S_Gain_Init
                        .dw  I_P_S_Gain_Block
                        .dw  I_P_S_Gain_Bkgnd

I_P_log10_to_log2        .dw  .f2b(.log2(10)/(20*2))
```

```
#####
# Common Code
#####

.code_cc

#####

I_P_S_Gain_Init
    i7 = i7 + (6)
    a0=0
                                ymem[i7] = a0
    ret

#####

I_P_S_Gain_Block
    mr_sr = (0)

# get public (i6) and private (i7) data
    i0 = ymem[i7]; i7+=1                # check enable/disable
    i5 = ymem[i7]; i7+=1                # i5: Input
    a0 = ymem[i0]
    a0&a0
    if (a==0) jmp >end

    nm5 = ymem[i7]; i7+=1
    i0 = ymem[i7]; i7+=1                # i0: Output
    nm0 = ymem[i7]; i7+=1
    i4 = ymem[i7]; i7+=1                # i4: Blocksize
    y0 = ymem[i7]; i7+=1

# indirect I/O pointers from OS
    i5 = ymem[i5]
    i0 = ymem[i0]

# Do the Dew...
                                x0 = ymem[i5]; i5+=n
    do ( i4 ), >
        a0 = x0*(unsigned)y0; x0 = ymem[i5]; i5+=n
%                                ymem[i0] = a0; i0+=n

%end
    ret

#####

I_P_S_Gain_Bkgnd
# translate public log gain to private linear gain
                                x0 = ymem[i6]                # y0: log gain
                                i7 = i7 + (6)
    call I_S_20log10_to_Linear
                                ymem[i7] = a0h                # a0h: linear gain

    ret

#####
```

```
# input:  x0 (max: +6dB (0x06000000))
#          (anything less than --90dB (0xa600) == -infdB)
# output: a0
#####

I_S_20log10_to_Linear
# convert from 20*log10 to log2
# and move from 8.24 to 9.23 (extra shift in log10_to_log2 constant)
  a0 = (0x0f80)                # bias == 32-1
  y0 = ymem[I_P_log10_to_log2]
  a0 += x0*y0

                                x0 = a0

# convert back to linear (2^x)
  logexp X=exp(x0)  Y=sm(x0)
  nop
  x0,y0 = logexp
  a0 = -x0*y0
  a0 = a0 << 1

ret

#####
#
#   MIPS: BlockRate*(19 + (2*block_size))
#
#   Example: for Blocksize = 16, Fs = 48kHz
#
#       BlockRate = 48000/16 = 3000 calls/second
#       MIPS = 3000*(19 + (2*16)) = 153,000 = .153 MIPS
#
#   Note: Bkgnd MIPS are ignored (assuming execution as a background task)
#
#####
```

Appendix B. *channel_select.pres.xml* Example

Presentation XML file for the *channel_select* primitive built in the tutorial section.

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2001 - 2004 Peak Audio, a division of Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<ningaloo_document xmlns:xpp="www.peakaudio.com/xmlns/xmlpreprocessor"
  xmlns:xpr="www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator">
  <device_type_presentation>
    <!-- device type identifier - the .pres.xml, .if.xml, and .imp.xml
      files must all have the same values for these elements -->
    <device_type_identifier>
      <family value="coyote_dsp"/>
      <type value="channel_select"/>
    </device_type_identifier>
    <!-- include commonly used definitions for device and runtime
      control presentation -->
    <xpp:include file="coyote_dsp_device.presinc.xml"/>
    <xpp:include file="ctl_args_stock.presinc.xml" />
    <xpp:include file="ctl_styles.presinc.xml" />
    <!-- device args - this section defines the basic appearance of device -->
    <xpp:insert name='device_args_router' /> <!-- this is a "router" device -->
    <!-- control panel - this section defines the layout of the
      runtime controls for the device -->
    <xpp:define name="control_panel">
      <column>
        <xpp:define name="ctl_args" >
          <member name='text' value='Select' />
          <member name='hover_text' value='Channel' />
          <member name='ruid' value='select' match='true'/>
          <member name='color_rgb' value='255 200 180' rule='color_rgb' once='true' />
          <xpp:evaluate expression="ctl_options=''" />
          <xpp:repeat index="c" first="1" last="input_count" >
            <xpp:evaluate expression="ctl_options=ctl_options+c+':'" />
          </xpp:repeat>
          <member name='ctl_options' xpr:value='ctl_options' />
        </xpp:define>
        <xpp:insert name="ctl_combobox_with_label" />
      </column>
    </xpp:define>
    <!-- device layout - this section defines the layout of the GUI components
      of the device, including audio and control signal ports -->
    <xpp:define name="device_layout">
      <!-- define input control signals -->
      <xpp:define name="device_input_control_signals">
        <xpp:evaluate expression="input_control_signals='&quot;enable&quot;'" />
      </xpp:define>
      <!-- and x x 1 audio in/out -->
      <xpp:insert name="coyote_dsp_x_by_1_presentation"/>
    </xpp:define>
    <xpp:insert name="device_instantiation"/>
  </device_type_presentation>
</ningaloo_document>
```


Appendix C. *channel_select.if.xml* Example

Interface XML file for the *channel_select* primitive built in the tutorial section.

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2005 Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<!--$Header: /home/cvs/eng/CS49400/Tools/CoyoteCAD/devices/gain.if.xml,v 1.4 2005/01/20 22:39:16 jmcooper
-->
<ningaloo_document xmlns:xpp="www.peakaudio.com/xmlns/xmlpreprocessor"
xmlns:xpr="www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator">
  <device_type_interface>
    <!--TYPE INFORMATION-->
    <device_type_info>
      <device_type_identfier>
        <family value="coyote_dsp"/>
        <type value="channel_select"/>
      </device_type_identfier>
      <device_info>
        <manufacturer>Cirrus Logic Inc.</manufacturer>
        <manufacturer_short>Cirrus Logic</manufacturer_short>
        <manufacturer_url>http://www.cirrus.com/</manufacturer_url>
        <model_short>Channel MUX</model_short>
        <model_name>Channel MUX</model_name>
        <model_number>Gain</model_number>
        <model_description>Gain</model_description>
        <model_url>http://www.cirrus.com/</model_url>
      </device_info>
    </device_type_info>
    <!--PROPERTY SCHEMA-->
    <device_property_schema>
      <property type="integer" default="2" label="Channel count" ruid="input_count">
        <restriction>
          <min value="1"/>
          <max value="16"/>
        </restriction>
        <help>channel count</help>
      </property>
    </device_property_schema>
    <!--CONFIGURABLE DATA SCHEMA-->
    <device_data_schema>
      <!--controls schema-->
      <controls>
        <control ruid="select" type="enum" default="1" label="Channel">
          <restriction>
            <xpp:repeat index="i" first="1" last="input_count" >
              <enum xpr:name="'Channel '+i" xpr:value="i" />
            </xpp:repeat>
          </restriction>
          <help>Selected input channel</help>
        </control>
      </controls>
      <!--ports schema-->
      <ports>
        <port ruid="output" domain="coyote_dsp_audio" type="source" label="channel output" />
        <port ruid="input" domain="coyote_dsp_audio" type="sink" label="channel input" />
        <xpp:repeat index="i" first="1" last="input_count">
          <port xpr:ruid="'input_' + i" domain="coyote_dsp_audio"
            type="sink" xpr:label="'audio channel(' + i + ')'"
            xpr:index="i-1" />
        </xpp:repeat>
        <port ruid="enable" domain="coyote_dsp_control_signal" type="sink" label="enable" />
      </ports>
    </device_data_schema>
  </device_type_interface>
</ningaloo_document>
```

Appendix D. *channel_select.imp.xml* Example

The following example on pages 42-43 shows the implementation XML file for the *channel_select* primitive built in the tutorial section.

```
<?xml version="1.0" standalone="yes"?>
<!--Cirrus Logic Confidential Information.-->
<!--Copyright (C) 2005 Cirrus Logic, Inc.-->
<!--All rights reserved.-->
<!--$Header: /home/cvs/eng/CS49400/Tools/CoyoteCAD/devices/gain.imp.xml,v 1.9 2005/02/03 23:21:01 jhilbert
$-->
<ningaloo_document xmlns:xpp="www.peakaudio.com/xmlns/xmlpreprocessor"
                  xmlns:xpr="www.peakaudio.com/xmlns/xmlpreprocessor/expressionevaluator"
                  xmlns:sxpr="www.peakaudio.com/xmlns/strudel/expressionevaluator">

  <device_type_implementation>
    <!--device type identifier-->
    <device_type_identifier>
      <family value="coyote_dsp"/>
      <type value="channel_select"/>
    </device_type_identifier>

    <!-- predefined module information -->
    <MCTsymbol value="X_P_BY_channel_select_PCT" />
    <MIPS value="((sample_rate/block_size)*(22+(2*block_size)))/1000000.0"/>
    <objectFiles>
      <file value="./channel_select.0"/>
    </objectFiles>

    <!-- default control port buffer assignments -->
    <ports>
      <port ruid="enable" buffer="I_VY_one" />
    </ports>

    <!-- data structures -->
    <structures>
      <!-- public instance data -->
      <structure>
        <name>public_data</name>
        <member>
          <name>select</name>
          <type>integer</type>
          <!-- the following element indicates that this data member is
              associated with an external controllable variable called "select" -->
          <control_var>select</control_var>
          <access>write</access> <!-- read, write, or both -->
          <initialize>1</initialize>
        </member>
      </structure>

      <!-- private instance data -->
      <structure>
        <name>private_data</name>
        <member>
          <name>private_y_data</name>
          <type>private_y_data_t</type>
          <memory>Y</memory>
        </member>
      </structure>
      <structure>
        <name>private_y_data_t</name>
        <member>
          <name>enable_ptr</name>
          <type>bool*</type>
          <port>enable</port>
        </member>
        <member>
          <name>max_channel</name>
          <type>integer</type>
          <sxpr:initialize>input_count</sxpr:initialize>
          <comment>input channel count</comment>
        </member>
      </structure>
    </structures>
  </device_type_implementation>
</ningaloo_document>
```

```

    <member>
      <name>output_ptr</name>
      <type>integer**</type>
      <comment>output buffer ptr</comment>
      <port>output</port>
    </member>
    <member>
      <name>output_stride</name>
      <type>integer</type>
      <comment>output stride</comment>
      <port_stride>output</port_stride>
    </member>
    <member>
      <name>input_ptrs</name>
      <type>input_ptr_t</type>
      <dimension>input_count</dimension>
    </member>
  </structure>
</structure>
  <name>input_ptr_t</name>
  <member>
    <sxpr:name>'input_ptr'+(i+1)</sxpr:name>
    <type>integer**</type>
    <comment>input buffer ptr</comment>
    <sxpr:port>'input_'+(i+1)</sxpr:port>
  </member>
  <member>
    <sxpr:name>'input_stride'+(i+1)</sxpr:name>
    <type>integer</type>
    <sxpr:port_stride>'input_'+(i+1)</sxpr:port_stride>
  </member>
</structure>
</structures>

<!-- include python scripts for translating external GUI control values to
      DSP control word values. NOTE: since the CDATA contents will be copied
      directly into a python script, and python is whitespace-sensitive, it
      is IMPERATIVE that the lines of python script between the CDATA begin/end
      start out NOT indented.
-->
<poke_crunch_function>
  <![CDATA[
# python function for crunching input control values
# nothing to do since control value == parameter value
#message.string_set( "select = " + str( select ) )
  ]]>
</poke_crunch_function>

</device_type_implementation>
</ningaloo_document>

```

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.

To find the one nearest to you go to www.cirrus.com

IMPORTANT NOTICE

Cirrus Logic, Inc. and its subsidiaries ("Cirrus") believe that the information contained in this document is accurate and reliable. However, the information is subject to change without notice and is provided "AS IS" without warranty of any kind (express or implied). Customers are advised to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, indemnification, and limitation of liability. No responsibility is assumed by Cirrus for the use of this information, including use of this information as the basis for manufacture or sale of any items, or for infringement of patents or other rights of third parties. This document is the property of Cirrus and by furnishing this information, Cirrus grants no license, express or implied under any patents, mask work rights, copyrights, trademarks, trade secrets or other intellectual property rights. Cirrus owns the copyrights associated with the information contained herein and gives consent for copies to be made of the information only for use within your organization with respect to Cirrus integrated circuits or other products of Cirrus. This consent does not extend to other copying such as copying for general distribution, advertising or promotional purposes, or for creating any work for resale.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). CIRRUS PRODUCTS ARE NOT DESIGNED, AUTHORIZED OR WARRANTED FOR USE IN AIRCRAFT SYSTEMS, MILITARY APPLICATIONS, PRODUCTS SURGICALLY IMPLANTED INTO THE BODY, AUTOMOTIVE SAFETY OR SECURITY DEVICES, LIFE SUPPORT PRODUCTS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF CIRRUS PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK AND CIRRUS DISCLAIMS AND MAKES NO WARRANTY, EXPRESS, STATUTORY OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE, WITH REGARD TO ANY CIRRUS PRODUCT THAT IS USED IN SUCH A MANNER. IF THE CUSTOMER OR CUSTOMER'S CUSTOMER USES OR PERMITS THE USE OF CIRRUS PRODUCTS IN CRITICAL APPLICATIONS, CUSTOMER AGREES, BY SUCH USE, TO FULLY INDEMNIFY CIRRUS, ITS OFFICERS, DIRECTORS, EMPLOYEES, DISTRIBUTORS AND OTHER AGENTS FROM ANY AND ALL LIABILITY, INCLUDING ATTORNEYS' FEES AND COSTS, THAT MAY RESULT FROM OR ARISE IN CONNECTION WITH THESE USES.

Cirrus Logic, Cirrus, the Cirrus Logic logo designs, DSP Conductor, and CobraNet are trademarks of Cirrus Logic, Inc. All other brand and product names in this document may be trademarks or service marks of their respective owners.