PurePath Studio

Driver Support

Header Files
Script Files
Script Loading Software

V1.2
7 June 2010

Table of Contents

1. Driver Architecture and Programmable Device Support

PurePath Studio (PPS) is the development environment for creating custom miniDSP programs known as Process Flows. The Process flow will contain static signal processing elements whose functions do not change during runtime, elements whose coefficients change based upon sample rate changes, and components that can be controlled via scripts during runtime. TI provides an example process flow with each of these component types.

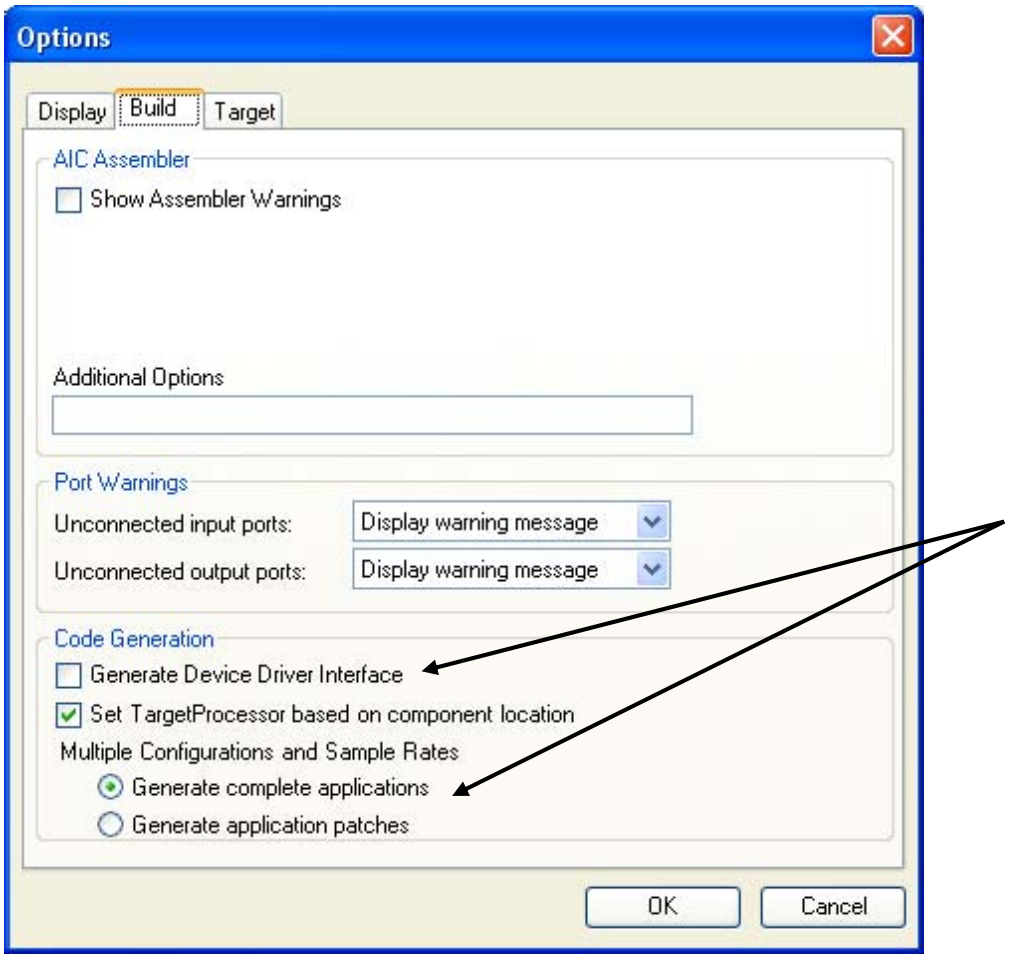
When a Process Flow is created and built in PPS, two outputs are created. The standard script format is for use with the PurePath EVM GUI or the Control Software. The second format is a “C” header file for building into a custom Device Driver. This header file can be used for a Linux or WinCE driver. Regardless of the format, both script files and header files contain miniDSP instructions, data and coefficients.

In addition, the “C” header file contains information about the volume and mux control locations within the miniDSP coefficients. This enables the Device Driver to create the proper audio controls and export them into the application environment so that these miniDSP controls are accessible by users and applications.

PPS version 2.0 and higher will produce a set of matching header (.h) and/or script (.cfg) files each time a Build is executed and the option is set requesting driver support. In the PPS Graphical Development Environment (GDE) this option is in the Tools menu, Build tab, Code Generation section. Check the “Generate Device Driver Interface” checkbox and then select the type of support required. The choices are “Generate complete applications” or “Generate application patches”.

The “Generate complete applications” option causes the cfg files to contain complete verbose settings of all registers required for the sample rate. The “Generate application patches” option causes the creation of cfg files that only contain the differences to switch from one sample rate to another. The patch files are smaller.

See figure below and/or PPS GDE documentation for additional information on Options.



Future versions of PPS will tag each file with a unique “Magic” number. The “Magic” number permits the Device Driver and the EVM GUI software to verify that the coefficients being loaded match the miniDSP instructions and memory map.

The Device Driver will perform a special AIC3254 control sequence during coefficient loading so that noise is not produced. This control sequence will include a mute of the outputs and possibly shutdown the miniDSPs.

Here is a diagram of how the various software components work together across separate environments to control and enhance the AIC3254’s programmable features. PPS also supports other programmable codecs with this same architecture.

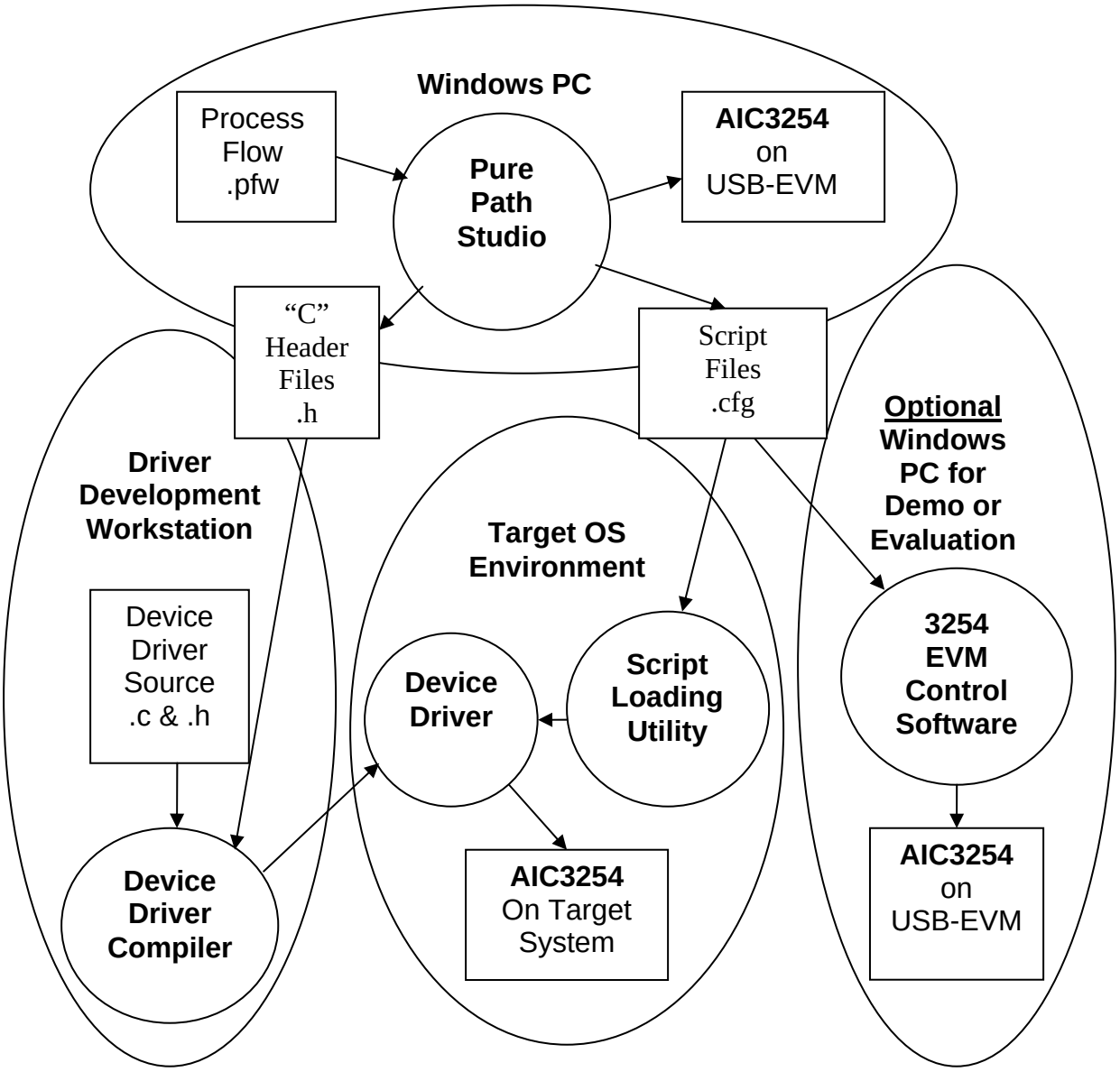


Diagram of how PPS and Device Driver work together to enhance the AIC3254

2. Support for compile time and run time

Support for compile time and run time interfaces to the PPS customized processes.

2.1 Driver Header File

The PPS header file is compiled into the device driver to support a custom process flow. The header is compiled into the driver to enable it to download the initial programmable configuration, specify the embedded runtime controls, and support other runtime modifications. The main "C" header file (.h) produced by PPS includes miniDSP instructions, data, and initial coefficient values. Also the main header can contain control structures describing the location of the coefficients for volume and multiplexer controls that exist within the miniDSP process.

These embedded controls for volume and mixer must be exposed into the application or user environment. Extended controls are created by the driver at initialization time for all of the volume and multiplexer controls specified in the main header file.

Additionally, if more than one sample rate is supported there will be additional header files and script files, produced by PPS, that contain the adjusted initial coefficient values necessary to support each sample rate. These can also be compiled into the driver so that the driver may support any available sample rate at initialization time.

2.2 Script File Loading Utility

Besides the "C" header files (.h), PPS will also produce Script files (.cfg) that contain sets of coefficients for making run time changes to the audio processing. Some may turn features on and off, while others may adjust Bi-Quad filters.

Often the system engineer will want to provide the capability to select between different "operational settings or profiles". These consist of one or more coefficient settings that specify different acoustic environments based on the base miniDSP process. For instance, there could be optional miniDSP processing sections within the PPS Process Flow, created by using mux's to enable or disable them. Changing the audio processing based on user selection, current input device, etc and greatly enhance the end-user experience.

Script files can be created in two formats. One is verbose and contains all register settings. The other is called a patch and only contains the differences to go from one sample rate to another.

A Script Load Utility will be provided for both Linux and WinCE environments to load these script files and communicate them to the Device Driver. The Device Driver will

optionally mute the codec outputs during the update to avoid producing audio artifacts for the components that do not have a soft update.

3. TI Supplied Drivers

3.1 Linux Driver Background

The default sound sub-system of Linux before kernel version 2.6 was Open Sound System (OSS). Beginning with Linux kernel 2.6 Advanced Linux Sound Architecture (ALSA) became the default sound interface. ALSA includes an OSS compatibility layer for older applications. Goals of the ALSA project:

- Automatic configuration of sound-card hardware
- Graceful handling of multiple sound devices in a system
- Hardware mixing of multiple channels
- Full-duplex operation
- Multiprocessor friendly, thread-safe code

ALSA System-On-Chip (ASoC) provides better ALSA support for embedded system processors and portable codecs.

Major features of ASoC are:

- Portability of codec driver between platforms and machines
- Processor manufacturers provide ASoC support in Board Support Package (BSP)
- Easy configuration of I2S interface by registration of capabilities
- Dynamic Audio Power Management (DAPM) keeps codec in minimum power state while reducing pops and clicks
- Separates the Device Driver into three components:
  - Codec Driver
    - Audio interface capabilities
    - Audio control functions
    - Codec I/O functions
    - DAPM functions
  - Platform Driver
    - Audio DMA engine
    - I2S and PCM audio data interface drivers
  - Machine Driver
    - Handles events to control other machine specific hardware like external amp's

3.2 Linux Driver and Software Features

- Basic audio playback and record of PCM encoded wave files
- Application API and user interface controls for hardware features and analog controls

  - Master Volume
  - Input Select (Multiplexer)
  - Microphone and amplifier controls
- Support for Linux command line audio applications
  - aplay plays audio files
  - arecord records audio files
  - amixer controls mixers, mux's and volumes
- Playback and Record at all codec supported sample rates
- amixer provides support for directly modifying codec registers

3.3 WinCE Driver Background

WinCE, also known as Windows CE and Windows Embedded CE is a trimmed down, real-time version of desktop Windows for embedded systems. WinCE audio driver models include:

- MDD/PDD – Windows CE 3.0
  - Model Device Driver (MDD) supplied by Microsoft implements streaming functions through audio Device Driver Service Interface (DDSI) functions
  - Platform Dependent Driver (PDD) supports codec and platform dependencies
- UAM (Unified Audio Model) – Windows CE 5.0 and higher
  - Only 1 audio device
  - Only 1 stream per device
  - Poor support for looping or streaming
- Wavedev2 – Windows CE 5.0 and higher
  - Combines UAM and MDD/PDD interface components
  - Multi-stream input and output
  - Built in mixing, volume, and format conversion
  - Automatic environment controls and stream routing

WinCE 5 and 6 have different memory access interface functions and user interface handling that makes some drivers incompatible across versions. TI drivers will support both version 5 and version6 via compile time switches.

3.4 WinCE 5/6 Wavedev2 Driver Features

- Wavedev2 driver model
- Support WinCE 5 and 6 through compile switches
- Basic playback and record of PCM encoded wave files
- Application API and user interface controls for hardware features and analog controls
  - Master Volume
  - Input Select (Multiplexer)
  - Microphone and amplifier controls

- User interface application required to support runtime script updates of coefficients
    - Reads PPS script files in standard format

4. Device Driver Header Generation

4.1 Overview

The GDE and related tools will be able to generate a header file for an embedded OS device driver such as Windows CE or Linux.

The header file format is listed below. The header file will be created cooperatively by the GDE and the ConfigBuilder application as follows:

- The GDE will generate a portion of the header related to components when the 'Generate Device Driver Interface Files' option is chosen in Tools/Options/Build Options. The GDE will provide the /driver option to the ConfigBuilder as well when this option is given.
- The ConfigBuilder will generate the program portion of the header if the /driver option is given.

The name of the header fill will be pps_driver.h and it will reside in the same directory as the generated .cfg file from PPS.

4.2 Header File Format

The header file format is listed below. The //CONTROL LOCATIONS section is generated by the GDE. The //INSTRUCTIONS & COEFFICIENTS section is generated by the ConfigBuilder.

```
//CONTROL LOCATIONS
typedef struct {
        u8 control_page;          //coefficient page location
        u8 control_base;          //coefficient base address within page
        u8 control_mute_flag;     //non-zero means muting required
        u8 control_string_index;  //string table index
} control;

static struct control MUX_controls[] = {
{page,base,1,0},
{page,base,1,1}
};

static string MUX_control_names[] = {
"Mux_1",
"Mux_2"
};
```

```
static struct control VOLUME_controls[] = {
{page,base,0,0},
{page,base,0,1}
};

static string VOLUME_control_names[] = {
"Volume_1",
"Volume_2"
};

//INSTRUCTIONS & COEFFICIENTS
typedef struct {
        u8 reg_off;
        u8 reg_val;
} reg_value;

static string REG_Section_names[] = {
"miniDSP_A_reg_values",
"miniDSP_D_reg_values"
};

struct reg_value REG_Section_program[] = {
{0,page},
{1,0x0},
{2,0x0},
.
255,<string table index>    // load the reg values from the referenced
                            // REG_Section_names array
.
255,<string table index>    // load the reg values from the referenced
                            // REG_Section_names array
.
{0,page},
{1,0x0},
{2,0x0},
{n,0)
};

struct reg_value miniDSP_A_reg_values[] = {
{0,page},
{1,0x0},
{2,0x0},
.
{n,0)
};
```

```
struct reg_value miniDSP_D_reg_values[] = {
{0,page},
{1,0x0},
{2,0x0},
.
{n,0)
};
```

4.2.1 Control Location Information

The GDE generates the //CONTROL LOCATIONS section when the GDE option Generate 'Device Driver Interface Files' is selected in the GDE options.

The information in the header file is as follows:

4.2.2 Structure definition

This structure is output verbatim:

```
//CONTROL LOCATIONS
typedef struct {
        u8 control_page;          //coefficient page location
        u8 control_base;          //coefficient base address within page
        u8 control_mute_flag;     //non-zero means muting required
        u8 control_string_index;  //string table index
} control;
```

4.2.3 List of Multiplexer controls

For each multiplexer control (Mono_Mux_TI_v1 or Stereo_Mux_TI_v1), a line is added to the MUX_controls[] and MUX_control_names[] sections.

The information listed is as follows:

- control_page – The I2C page of the start of the Component Interface
- control_base – The I2C register of the start of the Component Interface
- control_mute_flag – non-zero value means the control requires muting when changed
- control_string_index – the index of this control's name in the MUX_control_names[] section

Example:

```
static struct control MUX_controls[] = {
```

```
{page,base, 1,0},
{page,base, 1,1}
};

static string MUX_control_names[] = {
"Mux_1",
"Mux_2"
};
```

4.2.4 List of Volume Controls

For each volume control (Volume_TI_v1 or Volume_Lite_TI_v1), a line is added to the MUX_controls[] and MUX_control_names[] sections.

The information listed is as follows:

- control_page – The I2C page of the start of the Component Interface
- control_base – The I2C register of the start of the Component Interface
- control_mute_flag – non-zero value means the control requires muting when changed
- control_string_index – the index of this control's name in the MUX_control_names[] section

Example:

```
static struct control VOLUME_controls[] = {
{page,base,0,0},
{page,base,0,1}
};

static string VOLUME_control_names[] = {
"Volume_1",
"Volume_2"
};
```

Note: Volume controls contain two properties:

- *Volume, an integer value representing -110 to +6 dB*
- *VolumeCoefficient – a coefficient value*

The mapping from Volume to VolumeCoefficient is provided in a mapping file vollog.txt. It is assumed that the device driver will also have this mapping available.

4.2.5 Instructions and Coefficient Information

The section //INSTRUCTIONS & COEFFICIENTS is produced by the ConfigBuilder. The ConfigBuilder opens the pps_driver.h file that was produced by the GDE and appends the following information.

4.2.6 Header

The following header is produced statically:

```
//INSTRUCTIONS & COEFFICIENTS
typedef struct {
        u8 reg_off;
        u8 reg_val;
} reg_value;

static string REG_Section_names[] = {
"miniDSP_A_reg_values",
"miniDSP_D_reg_values"
};
```

4.2.7 Register Section

The following information is generated from the reg[][] section of the code. The tuples {n, m} are the I2C register and value from the I2C writes produced by the reg[][] section.

The special register 255 is used for the PROGRAM_ADC and PROGRAM_DAC commands. Those commands produce an entry with register 255 and a value of 0 or 1, corresponding to the string table index in REG_Section_names[] for the miniDSP_A or miniDSP_D.

Example:

```
struct reg_value REG_Section_program[] = {
{0,page},
{1,0x0},
{2,0x0},

<other reg values>

{255,0}         // load the code contained in miniDSP_A_reg_values array

<other reg values>

{255,1}         // load the code contained in miniDSP_D_reg_values array

<other reg values>
```

```
{0,page},
{1,0x0},
{2,0x0},
{n,0)
};
```

4.2.8 Code Section

The code section lists the I2C writes for every byte of the miniDSP_A and miniDSP_D generated code. This is the same contents as the .cfg file section produced by the PROGRAM_ADC and PROGRAM_DAC commands.

Example:

```
struct reg_value miniDSP_A_reg_values[] = {
{0,page},
{1,0x0},
{2,0x0},
.
{n,0)
};

struct reg_value miniDSP_D_reg_values[] = {
{0,page},
{1,0x0},
{2,0x0},
.
{n,0)
};
```

5. TI AIC Script Language

TI supports a common script language or syntax that is used to control EVM control software and runtime driver control. Script files contain data in text format to be sent to the codec's control interface. The scripting language is quite simple. Each line in a script file is one command. There is no provision for extending lines beyond one line. A line is terminated by a line feed (0x0a) or a carriage return and line feed (0x0d0a). Characters can be either upper or lower case as all are converted to lower case (except for the string text in the Break command).

Script language files can be created by users with text editors and by the PurePath Studio AICAsm assembler. The files are used as input by the codec EVM GUI's and the device drivers of supported operating systems. The codec EVM GUI's send the commands through a USB device to the codec's command port. The device drivers will send the command directly to the codec's command port.

The first character of a line is the command. Commands are:

- **#** Comment
- **r** Read from the serial control bus
- **w** Write to the serial control bus
- **>** Extend repeated write commands to lines below a **w**
- **i** Set interface bus to use (GUI only)
- **b** Break (GUI only)
- **d** Delay
- **f** Wait for Flag

Command Syntax

REGISTER WRITE and INCREMENT WRITE
w [i2c address] [register] [data_1] [data_2] ... [data_k]
> [data_k+1] ... [data_m]
> [data_m+1] ... [data_n]

Where 'i2c address', 'register' and 'data_x' are in hexadecimal format.

NOTE: n is less than or equal to 32 for a GUI communicating through the TAS1020B.

REGISTER READ
r [i2c address] [register] [read amount]

Where 'i2c address', 'register' and 'read amount' are in hexadecimal format and read amount is less than or equal to 32 (0x20).

COMMENT
[any text]

Lines beginning with the # character are ignored.

INTERFACE (EVM GUI ONLY)
i [interface]

Where 'interface' is:

i2cstd	Standard mode I2C Bus
i2cfast	Fast mode I2C bus
spi8	SPI bus with 8-bit register addressing
spi16	SPI bus with 16-bit register addressing
gpio	Use the USB-MODEVM GPIO capability

BREAKPOINT (EVM GUI ONLY)
b ["string"]

Where "string" can be any string of characters to be displayed at a pop-up message by the GUI.

DELAY
d [milliseconds]

Where 'milliseconds' is the delay time in decimal format.

WAIT FOR FLAG
f [i2c address] [register] [D7][D6][D5][D4][D3][D2][D1][D0]

Where 'i2c address' and 'register' are in hexadecimal format and 'D7' through 'D0' are in binary format with values of 0, 1 or X for don't care.

6. Script Load Software

6.1 Introduction

The purpose of this section is to describe the function of a software application for loading script files containing coefficients into the miniDSP register map through the device driver. The new programmable codecs require software support to enable their enhanced functionality to be fully utilized by customers. The Register Maps of these codecs contain the instruction and coefficient memory for the miniDSP's. The Device Driver will load the initial miniDSP instructions, data and coefficients into the respective Register Maps.

In addition, some customer's products will require multiple sets of coefficients to support varying environmental conditions, device usage, or end user choices. These customers will require a software utility to load these script files at runtime.

The Pure Path Studio (PPS) application builds customized process flows for the miniDSP's. The outputs of PPS are "C" header files and TI format script files. Each PPS process flow describes one or more miniDSP processes that consist of instructions, data, and initial coefficient values. In addition, PPS can produce multiple run-time settings (groups of coefficient changes) for each of the miniDSP components. The instructions, data and initial coefficients are output into "C" header files for compiling into a device driver. The run-time settings are output into TI format script files.

The script files will contain all of the necessary command and miniDSP register data to update the miniDSP memory. The Volume control component is "soft" (creates incremental changes when given a new target volume level). Changes to other components will require that the script contain Write commands to mute the outputs, disable the miniDSP, update the memory, re-enable the miniDSP and un-mute the outputs.

The standard name for the Script Load Utility is TiLoad.exe, but the name may be changed at any time by renaming the .exe file.

6.2 Script Load Utility

A software utility (an application with limited functionality) is needed to load the miniDSP script files produced by PPS and communicate them quickly and efficiently to the codec device driver which then performs a smart update of the coefficients in the codec's memory. The script files contain many commands that write data to the codec's Register Map.

The general functional phases of the script load utility are:
1) Handle specification of the script file path and name to load from the command line.

a) Optionally, support for a File Open common dialog would be a “nice to have” feature that would help in script development and debug. When no file name is specified on the command line a file open dialog would be displayed.

2) Load the PPS generated script file and parse the commands.
   a) The device driver, utility, and PPS must support the same common syntax for the format of the miniDSP command data in the script files.
   b) Parse the commands into contiguous groups of Writes, Reads, and other commands handled by the applications. Since groups of byte values following a single “w” command may be larger than a single coefficient, the data should be split into smaller groups of 32 bytes each.
   c) The commands are stored as binary data for transfer to the driver.
3) Save the current state of the driver.
4) Call the device driver interface that handles script loading.
   a) Each 32 byte buffer is then sent to the driver.
   b) The utility should either pass the Magic number from the PPS script file to the driver or query the driver for the current Magic number. This will allow the driver or utility to verify that the script matches the current set of miniDSP instructions and data.
   c) This interface should guarantee:
      i) The script buffer is executed immediately.
      ii) The script is loaded without any other codec activity occurring.
      iii) The codecs outputs are muted to prevent noise from intermediate coefficient value changes. This may not be necessary as the script may contain all controls sequences necessary to ensure proper and silent loading.
5) Execute Read, Delay, and Wait for Flag commands.
   a) Read and Wait for Flag require a mechanism to return data to the application from the driver.
      i) Wait for Flag implies a Read loop which tests the result. A timeout of 10 seconds should escape the loop.
      ii) Read should return data, but no action is required. This is a stub for customers to add custom actions.
      iii) Delay should wait the specified time by giving up the processor to the OS.
6) Restore the state of the driver.

6.3 “Smart” Functionality

Even though PPS will produce script files that include any necessary housekeeping operations, in some cases the utility will have to be “smart” and provide runtime analysis and correction. PPS will not know the current state of the codec and river, so the utility will have to analyze the requested configuration changes and perform additional functions as needed. “Smart” functionality includes saving the current driver state before other operations and restoring it afterwards. The audio driver may require additional interfaces to enable the utility perform these functions.

Some of the possible component coefficient changes should not take place while an audio stream is playing. The utility should analyze the script commands to know when to apply a script and when to display an error message that the script cannot be loaded.

6.4 Script Load Utility Syntax

TiLoad /F=File Path\Name
TiLoad /? or ? displays command syntax

Optional, nice to have features. RW and RR are redundant to amixer in Linux, but required in WinCE because there is no built-in command utility.

TiLoad with no command line arguments will display an open file dialog
TiLoad /RW8=0x0000 or <decimal 16 bit value of register and 8 bit value>
Same as amixer set RW8 dd
TiLoad /RW16=0x0000 or <decimal 24 bit value of register and 16 bit value>
Same as amixer set RW16 dddd
TiLoad /RW24=0x00000000 or <decimal 32 bit value of register and 24 bit value>
Same as amixer set RW24 dddddd
TiLoad /RR8=0x00 or <decimal 8 bit register to read> (displays returned 8 bit value)
Same as amixer get RR8 dd
TiLoad /RR16=0x00 or <decimal 8 bit register to read> (displays returned 16 bit value)
Same as amixer get RR16 dd
TiLoad /RR24=0x00 or <decimal 8 bit register to read> (displays returned 24 bit value)
Same as amixer get RR24 dd

6.5 TI Format Script Files

Script files must follow the previously described TI AIC Script Language format and rules.

6.6 Command Parsing

The command parser would read the script file and search for known commands. Anything else would be ignore and deleted until a valid command was found. Commands, as listed above, are only valid at the beginning of a line.

Comments will be ignored and not transferred to the communications buffer.

Write and Increment Write commands would be condensed into text a buffer (or encoded in binary?) and at the end or when non-Write command was encountered, send the buffer to the driver.

A Read command would fill a buffer with read commands until a non-Read command was encountered and send the Read buffer to the driver and wait for the return buffer. At this time the Read return data will be ignored, until custom code is added to respond to the Read values.

Delay commands would cause an application Wait or Delay of the appropriate time.

Wait for Flag command would cause the application to continuously (every 100ms) send Read commands to the driver until the flag value was satisfied in the returned data.

6.7 Utility Operating Environments

Linux includes a command line utility that can communicate individual register and value pairs to the device driver. While this is sufficient for simple changes and debugging, it does not handle loading large groups of coefficients at once.

WinCE does not have a comparable command line utility. A separate utility will be required to support command line register map changes and therefore this same utility will be used to load files of commands.

A utility program will be provided for the Linux and WinCE environments that handles the generic loading of scripts. Additionally the source for this utility will be distributed to any customer that would like to implement custom actions and script handling.

6.8 Linux ASoC Environment

The TI Script Load utility will have to work in steps. An initial step will query and store the current state of the ASoC driver. Then it will parse the input file and analyze the commands to determine if additional actions are required to facilitate the configuration changes. Each group of changes (write and increment write commands) are then sent to the driver in binary format. Any large groups of coefficients following a single “w” command should be split into smaller groups in individual buffers of 8 coefficients. (8 coefficients plus the starting register offset would be a 33 byte buffer.) Smaller Write commands should be processed individually as they may contain page changes.

Groups of writes in the script may be interspersed with reads and other commands which are processed in between the configuration change groups. Once the configuration changes are complete the utility will enter a restoration phase which will restore the ASoC driver to its original state.

To facilitate this communication between the audio driver and the Script Load utility, the audio driver will expose/export interfaces through the ASoC driver by which a user level application can verify/discover the state of the audio driver. This includes DAC and ADC and the corresponding miniDSP modes. Once this interface is invoked by the user

space application, the ASoC driver can then either stop the codec driver from running or at least cause an error message to the user that a playback/recording is in progress and configuration change cannot proceed. The ASoC driver will have to export interfaces for the following run-time states:

- Support for reading/restoring the DAC volume settings
- Support for reading/restoring the ADC gain/control settings

Depending on the ASOC driver model, the above may be simple or complex to implement. This may also mean that the ALSA/ASoC Library APIs which are available in the user space may have to be modified in order to support these custom interfaces.

6.9 WinCE Environment

On WinCE, the WaveAPI.dll already has interfaces to support custom property get/set actions. This is exactly the motivation behind the WaveOutSetProperty() and WaveOutGetProperty() Calls. However, the arguments to these APIs are completely specific to the underlying platform and the Audio Driver. Hence we are planning to exploit these APIs on WinCE to support the above functionality.

7. Multiple Configuration Support

The GDE supports multiple configurations of a process flow. All configurations use the same process flow graph containing the same components in the process flow, connected in the same way. However, each configuration has its own set of properties for every component.

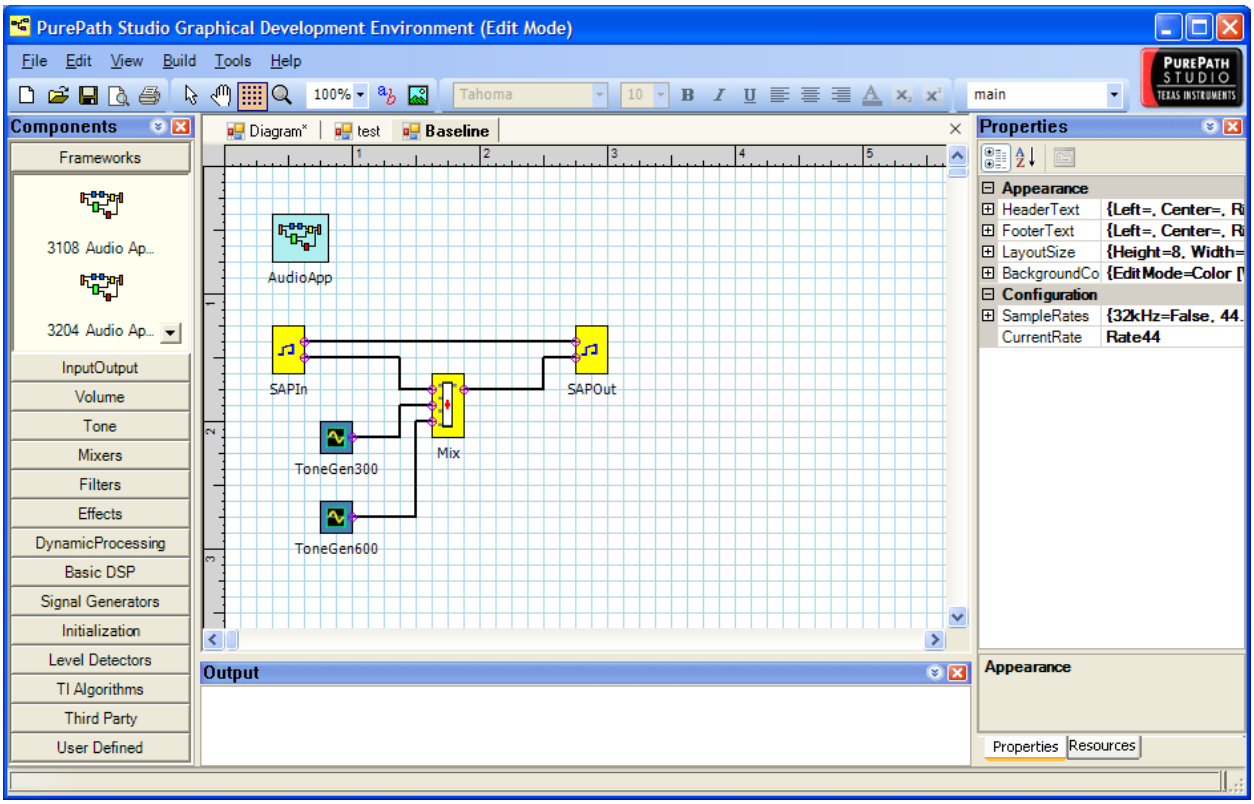
Configurations may be used to define a process flow with multiple behaviors, such as Jazz, Rock and Classical. As described below, configuration support is orthogonal to multiple sample rate support, which will be also reintroduced into the GDE at the same time.
Code Generation with multiple configurations and/or multiple sample rates will automatically build an image for each configuration and sample rate, as described below in Section 1.7.6.

7.1 Using Multiple Configurations

All new process flows start with one configuration, named **Main**. Process flows from previous versions of the GDE that did not support configurations will be converted on load to a process flow with one configuration, named **Main**.

The **Main** configuration must always exist, and cannot be removed or renamed by the user.

The current active configuration is visible in the Configuration pull-down list in the toolbar area of the GDE as
shown.



The user may add configurations using the Configuration Editor dialog available from the Tools menu.

Adding a new configuration to the process flow creates a new set of properties values for all those properties of all components that have not been marked as 'Shared between all configurations" as described in Section 1.7.3 below. The initial value of the new configuration's properties is the values from the current **Main** configuration.

The active configuration is shown in the GDE toolbar. The property window and the values of properties seen by pop-up GUIs are those properties from the active configuration.

7.2 Switching Configurations

Switching the active configuration is achieved by using the pull-down list on the GDE toolbar. In order to change configurations, no pop-up GUIs can be open. If any are open, the user is instructed to first close them.

Upon switching the active configuration, the Property Window is refreshed with the currently selected component's property values in the new configuration.

The active configuration can only be changed while in Edit mode. It cannot be changed in Run mode.

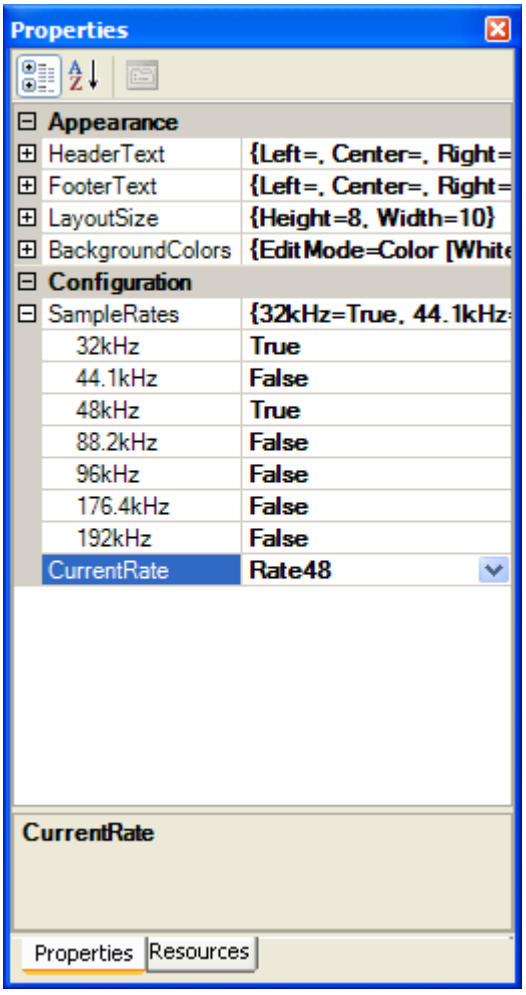
7.3 Shared Properties

Properties can have a unique value for each configuration, or the property value of the **Main** configuration can be shared by all configurations. Properties that are shared between all configurations can only be edited by the **Main** configuration.

All other configurations will display shared properties as read only in the property window.

7.4 User Interface Interaction with Configurations

The current configuration creates a 'view' of the current property set in the GDE. The Property Window displays the properties for components as they exist in that configuration. Only one configuration may be viewed at a time.



The diagram model properties SupportRates and CurrentRate that define multiple sample rate support are also available per-configuration. This allows one configuration (e.g. ‘DVD) to support one sample rate and another configuration (e.g. ‘MP3) to support a different sample rate, or multiple sampling rates.

Pop-up GUIs, process flow controllers, and the code generation macros such as %%prop() see only the property set for the current configuration in use. Code generation for multiple configurations is described in Section 1.7.6.

7.5 Multiple Sample Rate Support

Multiple sample rate support is orthogonal to configuration support. Each configuration may define a set of available sample rates.

In the GDE with multiple sample rate:

- Both Design-time and Run-time properties are allowed to be defined as rate-dependent.
- The SupportedRates and CurrentRate properties are available on a per-configuration basis.

With this combination of configurations and multiple sample rate, the GDE allows building of process flows with both different property values based on configuration and different property values based on sample rate, where components support sample rate dependent property values.

7.6 Code Generation with Configurations and Multiple Sample Rates

Generation of code in the GDE is performed based on the current configuration and sample rate, and the additional defined configurations and sample rates. The GDE can build multiple configuration applications two different ways. The choice is based on the ‘Multiple Configuration Build’ setting in the Tools/Options Build tab.

- If Generate Complete Applications is chosen, the GDE builds a complete application for each process flow configuration and sample rate as described below in Section 1.7.6.1.
- If Generate Application Patches is chosen, the GDE builds the current configuration and sample rate as a complete application and builds patches for all the configurations, including the current configuration as described in Section 1.7.6.2.

7.1.1 Code Generation for Complete Applications

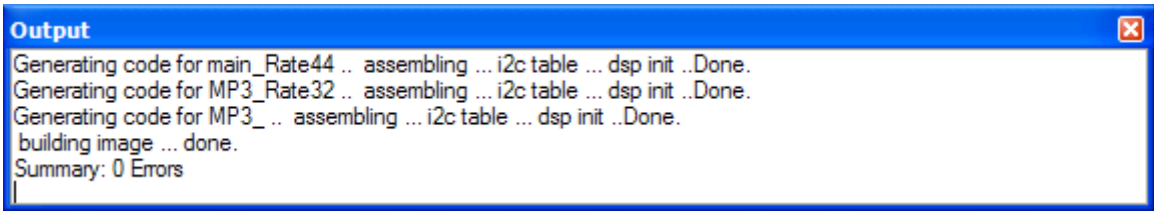
For each configuration and sample rate defined in the process flow, a complete application is built by the GDE. Code generation in the GDE occurs in subdirectories of

the directory where the .pfw file is located. Each subdirectory will contain the configuration name and the sample rate name for the configuration being generated. The applications are named with the configuration and sample rate name and are placed in the parent directory along with the .pfw file.

For example, consider a process flow with:

- A **Main** configuration with one process flow defined with one supported rate: 44.1Khz
- An **MP3** configuration defined with supported rates 32Khz and 48Khz.
- Code generation for this process flow would result in the following subdirectories containing generated assembly code from the GDE and the generated .image code from the assembler.
    - Main_Rate44\
    - MP3_Rate32\
    - MP3_Rate48\

The GDE output window will show the build for each configuration and sample rate.



This processing is outlined in the graphic below for the above example:
• A process flow is built in the GDE with two configurations (Main and MP3)
• The **Main** configuration has one sample rate (44.1Khz)
• The **MP3** configuration has two sample rates (32Khz and 48Khz)

In each subdirectory, the GDE builds applications as .image files. Then the ConfigBuilder builds the application .cfg files in the same parent directory as the .pfw file. The ConfigBuilder also builds device driver header files (see Section 1.8) in the parent directory.

• Main_Rate44.cfg – contains the entire application for the Main_Rate44 configuration
• MP3_Rate32.cfg – contains the entire application for the MP3_Rate32 configuration
• MP3_Rate48. – contains the entire application for the MP3_Rate48 configuration

7.1.1 Code Generation for Application Patches

If the GDE option ‘Generate application patches’ is defined in the Tools/Options Build tab, in addition to building the complete application as described in 1.7.6.1 above, the GDE builds application patches that are capable of producing any of the configurations starting from any other configuration.

The ConfigBuilder tool processes the application images and performs an N-way difference detection. The resulting .cfg and .h files contain the application and coefficient code necessary to effect the change of the application to the given configuration.

The GDE is not able to download application patches at run-time. These application patches are meant for host processors, which may need to reset and reconfigure the Portable Audio device prior to downloading.

Given the same example as above:

• A process flow is built in the GDE with two configurations (Main and MP3)
• The **Main** configuration has one sample rate (44.1Khz)
• The **MP3** configuration has two sample rates (32Khz and 48Khz)

The Main configuration is the current configuration and 44.1Khz is the current sample rate in the GDE in this example.

The GDE builds the current configuration and sample rate as a full application. Then, patches are built that define each configuration and sample rate, including the current one. These patches are the initialized program and data that are necessary to go from any other configuration to the given configuration.

7.1.1 Execution on the EVM

When Build/Download is chosen in the GDE, the GDE will download and execute the .cfg file corresponding to the currently selected active configuration and sample rate.

The active configuration and sample rate cannot be modified while in Run mode.

7.7 DEVICE DRIVER INTERFACE GENERATION

7.1.1 Driver Interface Overview

The GDE and related tools will be able to generate a header file for an embedded OS device driver such as Windows CE or Linux.

The header file format is listed below. The header file will be created cooperatively by the GDE and the ConfigBuilder application as follows:

- The GDE will generate a portion of the header related to components when the ‘Generate Device Driver Interface Files’ option is chosen in Tools/Options/Build

Options. The GDE will provide the /driver option to the ConfigBuilder as well when this option is given.

- The ConfigBuilder will generate the program portion of the header if the /driver option is given.

The header files will be generated differently based on whether the GDE is generating complete applications for each configuration, as described in Section 1.7.6.1, or generating patches for the non-current configurations, as described in Section 1.7.6.2.

The name of the header file will be ConfigurationName_Rate_pps_driver.h for complete applications and Patch_ConfigurationName_Rate_pps_driver.h for patches. The header file will reside in the parent directory, along with the .pfw file.

For the Patch_ConfigurationName_Rate_pps_driver.h files, only the //INSTRUCTIONS & COEFFICIENTS section is produced and will contain only the differences, just as it does in the corresponding .cfg files. The //CONTROL LOCATIONS section will be the same as the ConfigurationName_Rate_pps_driver.h file and will not be included in each of the patch header files.