A New Platform for the Development and Use of Real Time Audio Processing Applications

Soundart Development Team
support@soundart-hot.com

The appearance of the Chameleon system gives Digital Audio applications developers a fast, versatile and robust platform for the implementation and distribution of their algorithms. Final users take benefit too, by having a single equipment to run different applications depending on their needs on each moment. This article reviews the main features of the system and examines the basics of developing applications on this new platform.

Introduction

In recent years the application of Digital Signal Processing techniques to the music technology has experienced a great expansion, increasing the number of audio and music production products based on this technology.

The evolution of filtering and digital processing has given birth to a great number of dedicated hardware products and plug-ins designed to create sound effects. Those effects reach different audio fields such as filtering, modulation, delay lines, non lineal processing, frequency-time changes, auralization, spectral proccessing and others. Similarly, the development of different synthesis techniques has led to the market appearance of several synthesizers (hardware and software), with different features and results depending on the implemented algorithm chosen. Finally, the unstoppable evolution of personal computers has resulted in the availability of many pro and semi-pro computer based recording systems, with constant improvements in quality and efficiency. Those systems allow the integration of all kind of processors and synths as plug-ins or as stand alone applications, all of them unimaginable just a few years ago.

The Chameleon (Figure 1) has been created in this environment, designed to address and overcome the shortcomings of existing audio tools. The Chameleon is based around a Digital Signal Processor (DSP), it is optimized for real time processing of audio and MIDI signals, and can be programmed in unlimited configurations, by downloading compiled code via a PC host interface. By virtue of its own design philosophy, there are two different kind of users: programmers and final users. In one hand, there are DSP based applications programmers, who can develop, download and debug their applications with full access to all the Chameleon's hardware resources making use of all the necessary development tools. Once an application is complete, it may be easily distributed to run on another Chameleon. In this way, final users can benefit from a large and evolving pool of compatible applications without needing any programming knowledge, just selecting the application in function of their needs.

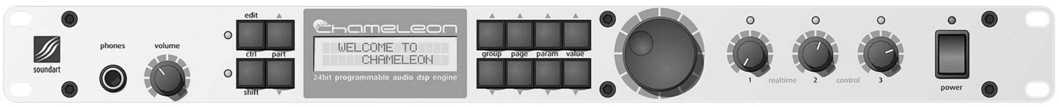

Figure 1. *As its name states, Chameleon adapts itself to the environment.*

Versatile and Efficient

The main idea behind the Chameleon is versatility and efficiency. The Chameleon doesn't process sound in any predefined way, it doesn’t do anything but can become any kind of "thing" that can be programmed on a DSP: from the most simple synthesizer to new and unusual sound effects. The character of the sound is entirely dependent on the algorithms chosen by the developer.

Existing DSP based devices are limited in their functionality. They do implement a complex real time algorithm efficiently, and some of them can be improved by software updates offered by the manufacturer, but their main function can't be modified. Updates may expand the device's features, improve performance or correct a bug introduced by a previous version. However, no device until now has offered users the facility to substantially change the behaviour or the sound of the underlying software.

Modern CPUs can execute extensive and sophisticated audio processes. But executing code in real time and developing critical In/Out time functions, often exceeds the capabilities of the operating system. Combinations of applications that work well individually can often crash the system when they run together, and the real-time stream management and processing is often compromised by quite unrelated system overhead. Individual variations in system configuration can also have drastic and unpredictable effects on audio processing.

All these limitations are overcome by the Chameleon, without sacrificing the portability and compatibility offered by computers, and the advantage of real time proccessing.

Architecture

The main component in the Chameleon is the Motorola DSP56303, from the DSP56000 fixed point CMOS family. This processor was chosen for its high processing speed and the functionality that it provides, which is also why it is present in many other DSP-based audio devices. The performance and programmer-friendliness of this chip family has established it as the standard for industrial audio processing: the DSP56000 family is employed in a majority of the DSP based pro audio products on the market today.

The DSP runs on a 100MHz clock, offering up to 100MIPS (Million of Instructions Per Second) with a 24 bit word length. Apart from features common to all DSPs, such as parallel MAC, delays, efficient address generation, circular adressing, etc. the 56303 offers improvements in processing efficiency like hardware nested controlled loops, quick interrupt return (minimizing latency), an on-chip DMA controller, two 56 bits accumulator registers and three internal busses, one for the memory and two others for data. This is an extension of the Harvard structure that allows to access two operands and one instruction in a single cycle. As well as the internal SRAM of the chip (8k words), the Chameleon board includes additional 4M words of DRAM.

Absolutely all the DSP resources are available to the programmer through the use of the provided development tools, as described below.

The Chameleon features two mono analog audio inputs and two mono analog audio outputs, (plus a stereo headphones out), connected to the DSP through a high quality AD/DA codec, which works with a 24-bit word length and a sampling rate of 48 KHz.

The management of all application tasks is done by a Motorola Coldfire MCF5206e microcontroller. It is a 32 bit processor running at 40MHz, with 8M bytes of external DRAM.

This microcontroller runs a real time multitask operating system, RTEMS, which manages the system resources – including the front panel - and offers a high level, driver-based, interface to the programmer. The MIDI I/O ports also connect to the Coldfire, which communicates and processes MIDI data using MIDIshare, another real time operating system designed specifically for developing musical applications. Applications are loaded and debugged by connecting the Chameleon to PC using a standard RS-232 interface.

For non-volatile storage of programs and parameters, there is 1M byte of Flash memory available, which is also managed by the Coldfire by using its appropiate RTEMS driver.

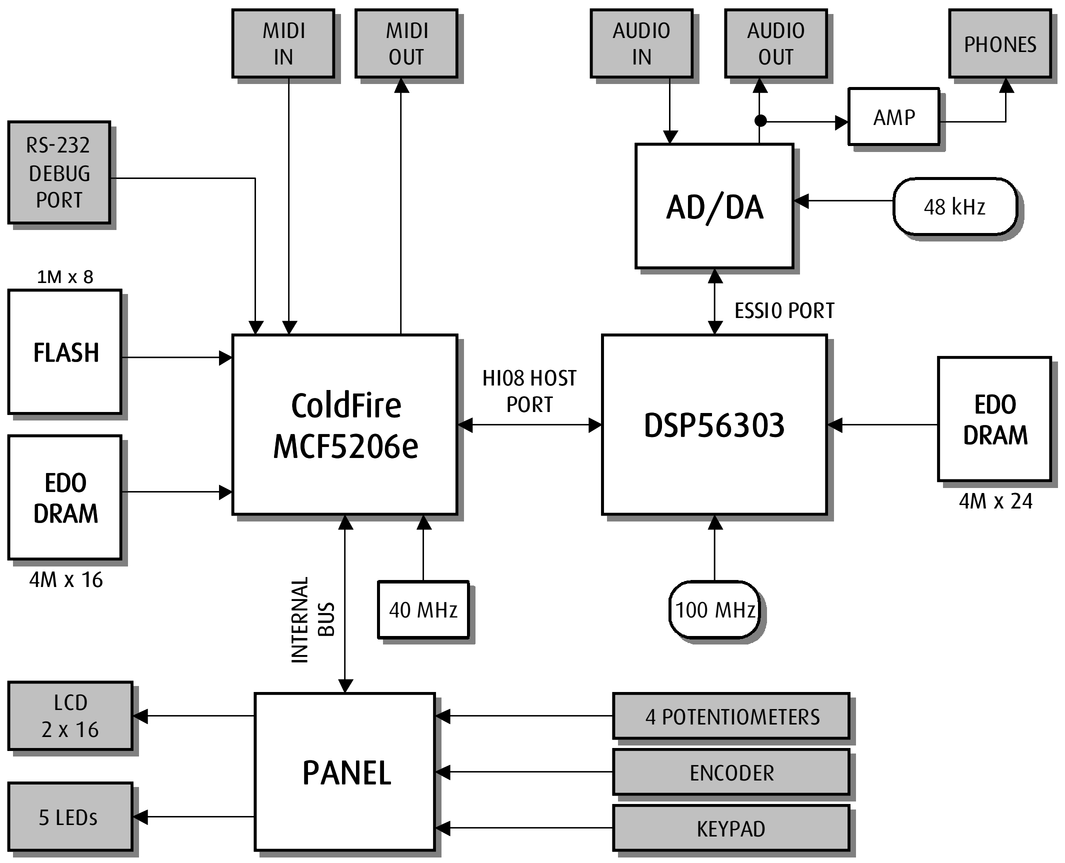

Figure 2: *Chameleon architecture*

Figure 2 shows a block diagram with the functional blocks of the Chameleon's hardware architecture.

The user can control application parameters with the generic controls of the front panel. There are 4 potentiometers, 1 rotary encoder, 12 keys and a 16x2 characters LCD display. The functionality and behaviour of individual controls is entirely up to the programmer. They can be used to change filter parameters, to save o recover presets, etc. When any of these controls is operated, the operating system notifies the event to the user program, which will act consecuently. In a similar way, the programmer can use the LCD to display the current state of the application or just to show messages to the world.

The ColdFire and the DSP are connected together via the DSP HI8 host port, which is a full-duplex 8 bit wide parallel and bidirectional port (although the data transfers between both procesors are done with 24 bit wide words in a transparent way).

Programming

Many real time audio processing algorithms can be implemented on general purpose DSP development boards. Most of the DSP manufacturers supply evaluation boards to test and get started with the DSP. The difference between these systems and the Chameleon is that this one is a complete system, wich already has all the needed peripherals. Thus, applications developed with the Chameleon are not mere prototypes, but complete and functional applications ready to be distributed and used in other units. This difference with other deveolpment systems is the essential characteristic of the Chameleon and shows it to be a specially useful tool. In fact, many applications developed on evaluation boards can be easily ported to the Chameleon, with the only addition of control routines to handle user interaction.

To make the applications development process easier, the programmer has available a complete Software Development Kit (SDK) that includes all the needed tools and code. The complete SDK is available for downolad at the Soundart's website www.soundart-hot.com. It consists of:

- Motorola Suite56TM Software Developing Tools, to compile, link, simulate and debug the DSP code.
- GNU C/C++ closs-platform compiler, with the development tools for the Coldfire.
- Compiled libraries containing the RTEMS and MIDIShare operating systems for the Coldfire.
- Libraries to access and control the hardware resources in the Chameleon.
- Tools to communicate a PC with the Chameleon, allowing to download and debug programs, and to generate distributable applications.
- An integrated development environment to generate and compile applications in a project orientated fashion.
- Extensive documentation including reference manuals, tutorials and source code examples.

Development of a complete application is a stepped process. The first step is to develop the DSP algorithms for audio manipulation. They may be implemented in DSP56303 assembler for optimal performance, or C for ease and portability. All the audio processing code will run in the DSP. On the other hand, the control code for interaction with the front panel, MIDI, parameter changes, etc. is created in C or C++, and it will run in the ColdFire.

Once compiled and linked, the DSP code, which contains the audio processing algorithms, can be tested and debugged in a simulator. Once debugged, this code is converted to a standard C header file (*.h*), using one of the conversion tools provided with the SDK. This file is then included by the ColdFire source code, which will download it to the DSP and make it start to execute. The ColdFire C/C++ code is also compiled with the needed libraries (which provide access to the operating system routines and hardware resources, apart from to the C standard libraries) to obtain a Coldfire executable file. All these steps are done transparently when compiling an application from the Development Environment.

Once the executable file is generated, it can be downloaded directly to the Chameleon from the PC Serial Port using the Toolkit debugging application. This way it will be tested and debugged while it is running in the hardware. Once it is dowloaded, the device will automatically execute the application. During the implementation process, the programmer can add some debugging messages to his code, which will be shown at run time in the terminal window of the Toolkit in the PC, so allowing to obtain the state of variables, events, etc. During the debugging process, the applications can be dowloaded and run directly in RAM memory, so reducing the download time and avoiding to write the FLASH memory each time.

Finally, when the application is debugged and finished, the executable file wich contains all the program can be permanently stored in the FLASH memory and will be executed every time that the Chameleon is turned on. This application can also be converted to a distributable MIDI file to allow other Chameleon owners to use it.

Figure 3 shows the relationship between the different software components of a Chameleon application. In a typical Chameleon application, the ColdFire controls all the system. RTEMS runs in the ColdFire and gives the programmer full access to the resources and the system programming.

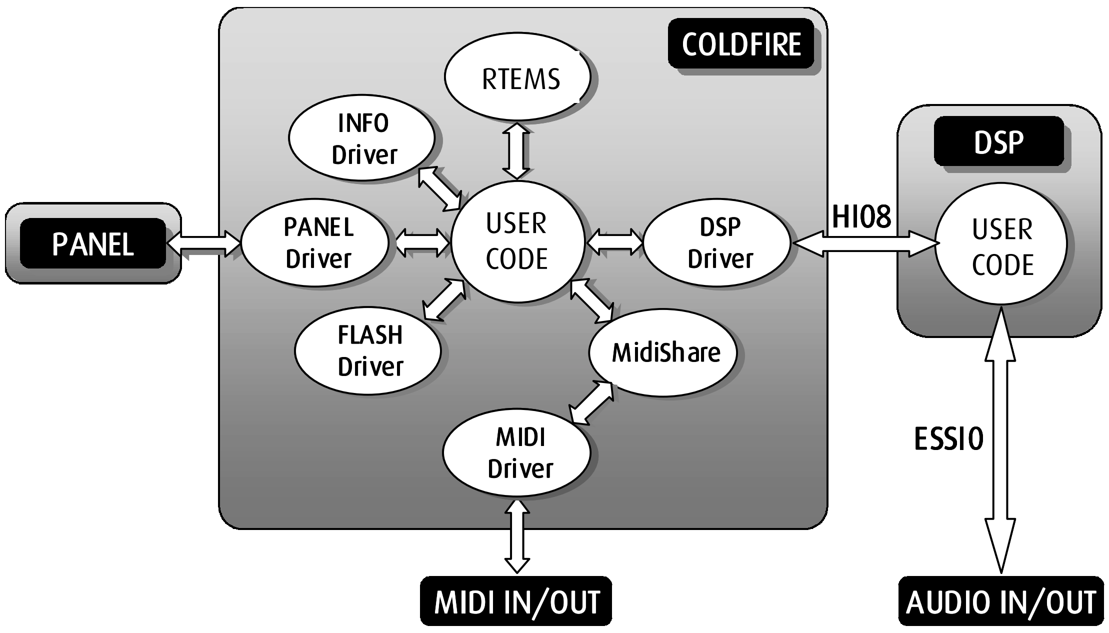

Figure 3: *Software blocks of a full Chameleon application*

RTEMS (Real-Time Executive for Multiprocessor Systems), developed by On-Line Applications Research Corporation and freely distributed, is a real time multitask operating system. It allows the programmer to create multiple tasks and provides the services needed to control devices, user code execution and a highly efficient embedded applications control environment (timers, semaphores, messages, events, dynamic memory managment, etc.).

Access to all system resources is done via device drivers, in such a way that there's a driver for each functional component of the board (FLASH memory, DSP, Control Panel and MIDI). An additional driver provides information about the version of the unit and the serial number (factory set in hardware), offering both forward compatibility and an optional anti-piracy security mechanism.

MIDIShare is provided to make easier the MIDI functionality programming. Developed by Grame and also freely distributed, MidiShare is a multitasking operating system designed for the management of MIDI applications. It is based on a client/server model and provides a powerful and efficient environment for developing MIDI applications, including temporization and synchronization control, communications, tasks and events, MIDI controllers, etc.

The DSP HI08 host port is used by both processors to exchange data. The ColdFire uses it to download the DSP code and to send control parameters or other application specific data when this code is running, and the DSP can send application specific data to the ColdFire.

Examples

Two easy sample applications are included to illustrate the typical scheme of a Chameleon application. The code listed for the two examples below is simplified for clarity purposes. The complete examples and documentation can be found in the Chameleon SDK.

The first example provided shows how to create basic code to access the front panel controls. This is the classic "Hello world" program, extended to add the framework to manage the user input through the front panel. A new task is created (the main one in this case), which initializes the front panel driver and scans it in a infinite loop looking for a new external events. Taking advantage of the multitask capability of RTEMS, other tasks could be created to run simultaneously. In this example no audio processing is done, so no DSP code has to be generated. Following is the listing of the resulting code.

```
/*  SoundArt 2002
 *  File: main.c
 *  Chameleon front panel control sample code
 */
#include <rtems.h>
#include <chameleon.h>

// Main task
rtems_task rtems_main(rtems_task_argument ignored)
{
       int                   panel;         // Panel's driver Handler
       rtems_unsigned8       potentiometer; // Id for the moved potentiometer
       rtems_unsigned8       value;         // Current value of the potentiometer
       rtems_unsigned32      key_bits;      // Codified value of the pressed key
       rtems_unsigned8       encoder;       // Id for the moved encoder
       rtems_signed8         increment;     // Increment/decrement encoder value

       // Open the Front Panel driver
       panel = panel_init();

       // Show a message on the LCD
       panel_out_lcd_print(panel, 1, 0, "Hello World     ");

       // Show a message in the debugger Toolkit terminal only in debug mode
       TRACE("Application Running\n");

       while (TRUE) // forever
       {
              // Wait for a new event
              panel_in_new_event(panel, TRUE));

              // Actions after receiving panel event
              if (panel_in_potentiometer(panel, &potentiometer, &value))
              {
                     // User has moved a potentiomenter. Add user's code here
              }
              else if (panel_in_keypad(panel, &key_bits))
              {
                     // User has pressed a key. Add user's code here
              }
              else if (panel_in_encoder(panel, &encoder, &increment))
              {
                     // User has moved a encoder. Add user's code here
              }
       }
}
```

This second example shows more in detail how a typical Chameleon application manages and controls the audio. This example implements an audio thru with a volume control. The DSP receives the incoming audio data stream from the audio inputs, multiplies it by a volume constant, and sends it back to the audio outputs. The volume is set by the user by moving a potentiometer in the front panel. Although this is a very simple example, it illustrates the framework to implement more complex audio processing applications.

The following listing corresponds to the code to be implemented in the DSP. After enabling its ESSI0 port, which enables the communication with the AD/DA codec and therefore the audio data receving and transmitting, the program enters an endless loop which checks if a new volume value has been received on the HI08 host port from the ColdFire, multiplies the audio input samples by this value and sends back the result to the audio output.

```
;*******************************************************************************
; Soundart 2002
; File: dspthru.asm
; Managing audio with the Chameleon. DSP audio processing code.
;*******************************************************************************

       include 'dsp_equ.asm'
Start:
       ;***********************************************************************
       ; Init Master Volume volume
       ;***********************************************************************
       MOVE    #<0,X0                       ; X0 register holds the master volume value

       ;***********************************************************************
       ; Enable ESSI0 transmit and receive (for digital audio input and output)
       ;***********************************************************************
       BSET    #CRB_TE0,X:<<CRB0            ; Enable Transmit
       BSET    #CRB_RE,X:<<CRB0             ; Enable Receive
       BRSET   #SSISR_RFS,X:<<SSISR0,*      ; Wait while receiving left frame
       BRCLR   #SSISR_RFS,X:<<SSISR0,*      ; Wait while receiving right frame

       ;***********************************************************************
       ; Main Loop start
       ;***********************************************************************

MainLoop:
       BRCLR   #HSR_HRDF,X:<<HSR,NoHostData ; Check if new data received on the HI08
                                            ; host port input
       MOVEP   X:<<HRX,X0                   ; if so, set new master volume value

NoHostData:
       ;******************************************************************
       ; Channel Loop, repeated twice, one timr for each stereo channel
       ;******************************************************************

       DO      #2,ChannelLoop
         BRCLR #SSISR_RDF,X:<<SSISR0,*      ; Wait until receive register is full
         MOVEP X:<<RX0,Y0                   ; Get new input sample in the Y0 register

         ;*************************************
         ; Processing code starts here
         ;*************************************

         MPY   X0,Y0,A                      ; Multiply the input sample by the volume
                                            ; and store it in the accumulator. A = X0*Y0

         ; Just insert user processing code here!
         ; *************************************

         BRCLR #SSISR_TDE,X:<<SSISR0,*      ; Wait until transmitter register is empty
         MOVEP A,X:<<TX00                   ; Send the result to the output

ChannelLoop:
       JMP     MainLoop                     ; Repeat forever

       end     Start
;*******************************************************************************
```

In the preceding code listing, it is important to note that almost no configuration tasks have to be performed in the DSP. The device's firmware already initializes all the DSP peripherals correctly at boot time, so the programmer only has to take care of enabling them and focus on the processing code itself. Taking the above skeleton, it would be extremely easy to start to implement a new processing algorithm, just by placing its specific code in the signaled processing code area.

Once the DSP code is implemented, it is compiled and the resulting DSP executable file is converted into a standard C header file by the appropiate conversion tool. In this case, this file will automatically named to "dspthru.h" (the same name as the main source file with a different extension). This header file contains the binary image of the DSP code, stored in a C array named *dspCode*. Thus, it can be accessed by the ColdFire to initialize the DSP and start its code to run. The continuing example below shows how this access is done.

Following is the code needed by the ColdFire to run the control part of the application. It will consists on two tasks or processes running simultaneously. The first one, *panel_task* will just scan the panel for new user inputs on any of the controls. In this example it will only respond to the volume potentiometer movement. Every time that this potentiometer is moved, a global value will be updated. The second task, named *dsp_task*, will periodically check if the master volumed has been modified by the panel task, and when this happens it will send the new volume value to the DSP through the host port (remember that the DSP code will then update this value to perform the actual volume change over the audio). It will use a rate monotonic service, which is a RTEMS service for performing accurate periodical tasks. A third task, the *rtems_main* task that will be executed only at the application startup, will start the other two tasks and finally will kill itself.

```
/*  Soundart 2002
 *  File: main.c
 *  Managing audio with the Chameleon. ColdFire control code.
 */

#include <math.h>
#include <rtems.h>
#include <chameleon.h>
#include "dspthru.h"  // binary image of dsp executable code converted to a header file

// Global variables
static rtems_signed32 master_volume = 0;     // current master volume value
static rtems_signed32 old_master_volume = 0; // old master volume to check value changes

/****** panel task *******/

static rtems_task panel_task(rtems_task_argument argument)
{
	rtems_unsigned8			potentiometer;
	rtems_unsigned8			value;
	int				panel;
	float				dB;	// auxiliary

	TRACE("Starting panel_task...\n");

	panel = panel_init();

	while (TRUE) // forever
	{
		panel_in_new_event(panel, TRUE);

		if (panel_in_potentiometer(panel, &potentiometer, &value))
		{
			switch (potentiometer)
			{
				case PANEL01_POT_VOLUME:
					//convert the pot value to dB in fixed point format
					dB = -90.0 + (float) 40.0*value/127.0;
					master_volume = float_to_fix(pow(10.0,dB/20.0));
```

```
                              // show the current volume value on the LCD display
                              sprintf(text, "   Volume %-3d   ", value);
                              panel_out_lcd_print(panel, 1, 0, text);
                              break;
                      }
              }
       }
}

/****** dsp task *******/

static rtems_task dsp_task(rtems_task_argument argument)
{
       rtems_id              period_id;     // id for the rate monotonic service
       int                   dsp;

       TRACE("Starting dsp_task...\n");

       // create a rate monotonic service, for the periodic execution of the loop below
       rtems_rate_monotonic_create(name, &period_id);

       // init the DSP driver, load the processing code into the DSP and start the code
       // execution
       dsp = dsp_init(1, dspCode);

       // infinite loop, each iteration is executed each 1 millisecond
       while (rtems_rate_monotonic_period(period_id, 1) != RTEMS_TIMEOUT)
       {
              // if the master volume has changed, update the volume in the DSP
              if (master_volume != old_master_volume)
              {
                     old_master_volume = master_volue;
                     // Send the new volume value to the DSP through the host port
                     dsp_write_data(dsp, &master_volume, 1);
              }
       }
}

/****** main task *******/

rtems_task rtems_main(rtems_task_argument ignored)
{
       create_dsp_task();    // create and start the DSP task
       create_panel_task();  // create and start the panel task
       rtems_task_delete(RTEMS_SELF); // kill itself and let the app running
}

/*-------------------------------------------------------------------------------*/
```

Fields of Application

Having in mind that the Chameleon is intended as multipurpose open audio DSP development platform, there exists a wide spectrum of applications where it can act as a useful development tool. It is also important to note that since it is also intended as a finished product, any serious software development made with the Chameleon can bring automatically a complete ready to market application.

For music oriented applications, implementation of efficient synthesis algorithms is possible, as well as to explore new trends and ways to obtain attractive sounding synthesis models. For audio effects processing, it is up to a developer’s imagination to find ways to explore and improve existing algorithms in artificial reverberation, frequency and amplitude modulation, distortion, physical modelling, device emulation, audio restoration, auralization, audio to midi conversion and so on.

In the field of Acoustics, it is also possible to develop measurement applications such as signal generation and analysis, with applications to audio measurement, architectural acoustics, vibrations measurements, speech recognition and generation, intelligibility evaluation and many others.

Even in other areas, non directly related to music or audio but still cathegorized into the low frequency signal processing field, the Chameleon can yeld exceptional results and offer ideal solutions with a relatively short development time. Some of those applications include mechanical and vibrational analysis, telecommunications, electromedical applications, sensors and control technologies.

Final Users

As stated above, the ultimate end of development on the Chameleon is the distribution of (mainly) musical applications to other users of the platform. Knowledge of programming and DSP theory are not required for the Chameleon end users to perform in studio, broadcast, home studios or stage environments who can fully exploit the unprecedented versatility and efficiency of this platform. Apart from the applications supplied with Chameleon, third party developers can sell or freely distribute their programs so they can be used by a wide users community.

Last but not least, the flexibility and features of the Chameleon allow the novice programmer to focus quickly on DSP technology. Combined with the reference-quality audio interface and the proven power of its DSP core, this makes the Chameleon an ideal system for the introduction and teaching of Signal Processing by universities and other educational centers.