soundart

TECHNICAL DOCUMENT
STD001

chameleon applications developer's guide

revision 1 | 05.2002
chameleon S.D.K. v1.2

Table of Contents

Chapter

0

About This Document

This manual is written for developers who wish to write audio and MIDI applications for the Chameleon. It is a guide for the Chameleon Software Development Kit (from now on Chameleon SDK), and it contains an introduction to all of the platform's hardware and software components, an introduction to the development tools provided by Soundart, and a guide to all the supplied documentation. This document covers the version 1.1 of the Chameleon SDK for the Chameleon model #01. To check for successive updates, availibility, software and news, it is recommended to visit Soundart's website www.soundart-hot.com.

Chapter

1 Introduction

Chameleon is a versatile device exclusively designed for the implementation of realtime audio and MIDI digital signals processing and synthesis. Conceived as an open platform, that can be totally programmed by the user, its specific functionality will depend on the code executed each time. This code can be generated by the user by way of the appropiate development tools, which are freely distributed and can be obtained free of charge.

The key concepts of the system are **control** and **processing**. The control side is carried out by means of a **front panel** programmable for the interaction with the user, and a 32 bit **microcontroller**, which carries out all the system's control tasks and communications. The processing part is done by means of a **digital signal processor** (DSP) which handles all the real time audio processing and/or generation.

The external device connections are two analog non balanced audio inputs and two outputs, a headphones stereo output, one MIDI input and one output, plus an standard RS-232 port for debugging purposes.

Chapter

2 Hardware Architecture

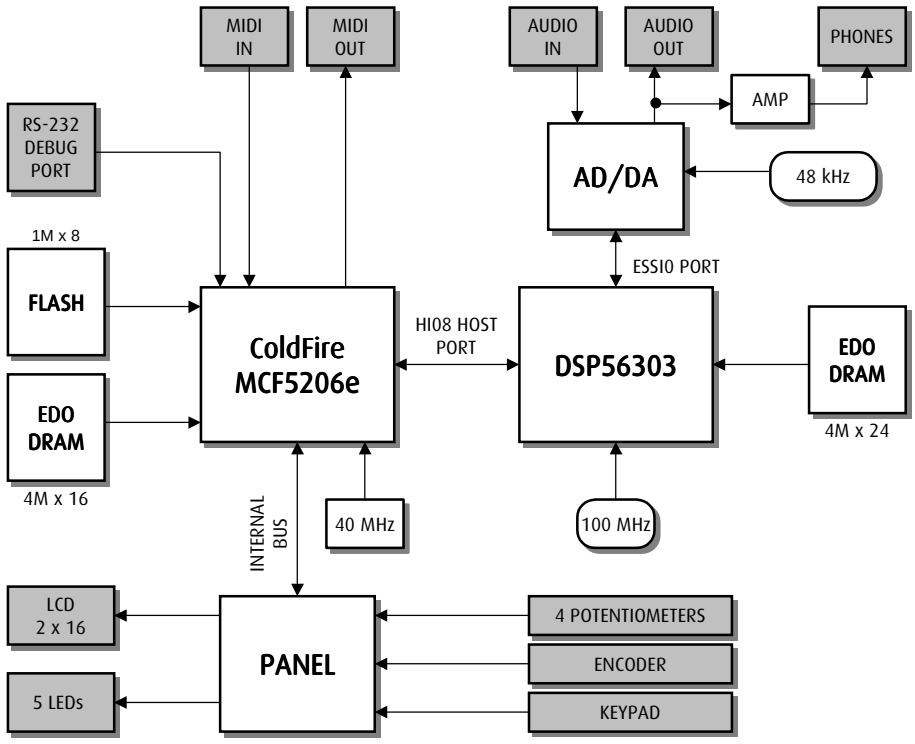
2.1 Introduction

In this section the Chameleon hardware architecture will be described, from the programmer's point of view. Before understanding the applications developing methodology, it is important to know the elements that integrate the device and to have a global idea about how these are related to each other in the platform's hardware structure.

Figure 2.1 shows the Chameleon functional blocks diagram. Internally it contains two processors working in a totally asynchronous way: a general purpose 32 bit Motorola's ColdFire family microcontroller running at 40 MHz, and a 24 bit Motorola DSP running at 100 MHz.

Figure 2.1

Chameleon Hardware Architecture



The DSP takes exclusively care of the audio generation and/or processing tasks. It has a connection to the analogic world through an AD/DA 24 bit stereo converter working at a sample rate of 48 KHz, so that it can receive digital audio from the AD converter's module, and to generate digital audio to the DA converter's module. The audio data are received and transmitted in synchronous interleaved serial format via the DSP's ESSI0 port.

Internally the DSP owns 8KWord[1] of high speed SRAM type memory. Furthermore, it has been endowed with a 4MWord DRAM EDO type external memory subsystem, totally available for the audio operations. The internal SRAM can be used to handle very frequently accessed data or routines. The external DRAM, which is slower, can be used to store less frequently used data or code, such as coefficient tables, delay lines, Host communication routines, etc. In any case, there is no restriction in the available memory use, except for terms of efficiency.

ColdFire is responsible for the handling all the system control tasks. In fact, the complete user's code will be downloaded on it, as we'll see in the following chapters. Following are the main tasks of such a microcontroller:

- System initialization.
- Downloading of the DSP code and sending the appropiate control signals and data during the application's execution.
- Non volatile memory handling.
- MIDI (event generation and reception) and RS-232 (debugging) communication control towards the exterior.
- Front panel controls management.

ColdFire owns an external memory subsystem composed by 8 MByte of DRAM EDO memory and 1 MByte of rewritable non volatile FLASH type memory for the permanent storing of data and code.

The front panel establishes the communication interface with the application's user through its controls. These controls are also handled by the ColdFire through an internal bus.

Although both processors, DSP and ColdFire, run independently, there exists a communication bus between the two which allows to perform data transfers in both directions. It is a 8 bit wide parallel bus, called HI08 Host Port, included in the DSP to handle direct connection with other microprocessors.

The next sections describe in further details all the above mentioned components.

[1] From now onwards, 1 Word = 24 bits except otherwise specified.

2.2 DSP

DSP is the main component of the Chameleon, which takes benefit from its huge signal processing power to allow powerful, efficient and complex applications implementation. Bearing in mind the objective of giving the programmer the whole DSP raw processing power, entire access with no restrictions to all its resources is provided, so the DSP can be exclusively dedicated to the application's specific processing tasks. Therefore the DSP programming is done without the use of any "intermediary" nor abstraction layers that could otherwise unnecessarily overload the application.

The flexibility obtained by complete access to the DSP has an evident drawback: for a Chameleon complete applications developer, an exhaustive knowledge of such component will be indispensable in order to obtain maximum benefit of its possibilities. This is the "price" that must be payed to become a true DSP programmer. For those who aren't familiar with the DSP, a comprehensive reading of the extensive documentation available is recommended. In addition to the documentation provided with the Chameleon SDK (see documentation index), it's also recommended to periodically visit the Motorola Semiconductors Products Sector webpage, www.mot-sps.com, to check for interesting up-to-date information.

The DSP used is the Motorola DSP56303,which belongs to the DSP56000 family of 24 bit fixed point digital signal processors, with modified Hardvard architecture optimized for multiply and accumulation (MAC) operations.

This DSP executes each instruction on a clock cycle, achieving 100 mega instructions per second (MIPS) with the 100 MHz clock that it is provided with. Its internal architecture allows a highly parallelizable instruction set (with up to an arithmetic operation, a logic one and two data moves in a single instruction), which includes hardware loop control to avoid the overhead inherent to the traditional software loop control instructions.

The Arithmetic-Logic unit operates with 24 bit-wide registers and uses internal 56 bit-wide accumulators to increase the dynamic range and so reduce the error accumulated during consecutive calculus. It has two different rounding methods (convergent and two's complement), saturation mode (without accumulator overflow), automatic scaling mode, and extended precission mode (double precission multiply).

The bus architecture divides the available system's memory into three types: P, X and Y. P memory space contains the program's instructions, and both X and Y memory spaces contain the data. With this structure, access to one instruction and two operands at the same time is achieved.

Memory access is highly optimized by means of a dedicated address generation unit, which allows flexible addressing modes. In addition to the linear data accessing, DSP has special accessing modes which are very useful for specific algorithm types, which do not imply overload on

the code: circular buffers and reverse ordering (the later is specially useful for the Fast Fourier Transform calculation).

The DSP owns 8 KWord of high speed SRAM internal memory (accessible without waiting cycles), which can be assigned in several ways (distributed into P, X or Y space memory), depending on the algorithm needs. It is also possible to use part of such memory to perform cache during the external memory accesses.

An important feature is the possibility to execute fast interrupt routines which avoid the typical overhead on the processor's interrupts. Once an interrupt source has been enabled and if the first two instructions in the interrupt's attention routine doesn't include a subroutine jump, the interrupt is executed in a "fast" way: only these two instructions are executed, without saving any registers onto the stack and with no need to include the return from interrupt instruction. This way of processing interrupts is very convenient in most cases and it notably minimizes the latency inherent to the concept of interrupt.

The DSP processor's core efficiency is boosted by a hardware stack structure, direct memory access (DMA) modules which can be used in several modes and automatic refresh generators for dynamic memories (DRAM).

2.3 ColdFire

As we shall see in the next chapter, since the whole control code that runs on the ColdFire can be created in high level language (C/C++) with operating system calls (RTEMS and MidiShare), the use of this microcontroller is transparent in a certain degree (and therefore so is the access to the system's resources), so that neither a deep knowledge about its internal architecture and operation nor its assembly language are not essential. Nevertheless, it is also recommended to read the provided documentation (see documentation index) and the periodic check for news and updates on the Motorola website (www.mot-sps.com) to achieve the maximum benefit from such powerful microcontroller.

The chosen microcontroler is a Motorola MCF5206e ColdFire. This processor uses a 32 bit word length and has a Varible-Length RISC instruction set (Generated code uses less memory than traditional RISC instrucion sets), which is a subset of those available in the Motorola 68000 family architecture, but using only one clock cycle per instruction instead.

ColdFire owns 8 KBytes of internal memory and a memory cache handler to reduce the waiting time during the external accesses. Additionally an external memory subsystem has been provided, comprising an 8 MByte of DRAM EDO memory plus 1 MByte of FLASH type non volatile and rewritable memory.

It also has two DMA channels, refresh generators to allow glueless DRAM memory connection, two serial communication ports and two 16 bit timers.

2.4 Front Panel

The front panel consist of the communication interface with the application user. Through this panel the commands are received and the current state of the running application is displayed.

Figure 2.2 shows the panel aspect with the most important components enlarged. It consists of:

- Four potentiometers. The rightmost three of them are enlarged on the picture, each with its own number, with the "realtime control" caption below all of them, as they can be assigned to control realtime parameters of the application. The fourth potentiomenter (placed on the panel's left) is marked as "volume", and it is conceived to act as the device's audio volume control although its actual functionality cand be changed arbitrarily.

- A programmable incremental rotative encoder, which can be used as another general purpose control.

- Twelve push buttons. These are divided into two groups, as showed on the figure. As their functionality is totally application dependent, their captions are specified in a generic way. These are conceived to act on a hierarchical menu tree structure, but is the programmer's concern to assing each of these twelve keys its specific function inside his application. The arrow above and below each button are intended to mean "up" and "down" (i.e. "param up", "param down", "group up", "group down" and so on).

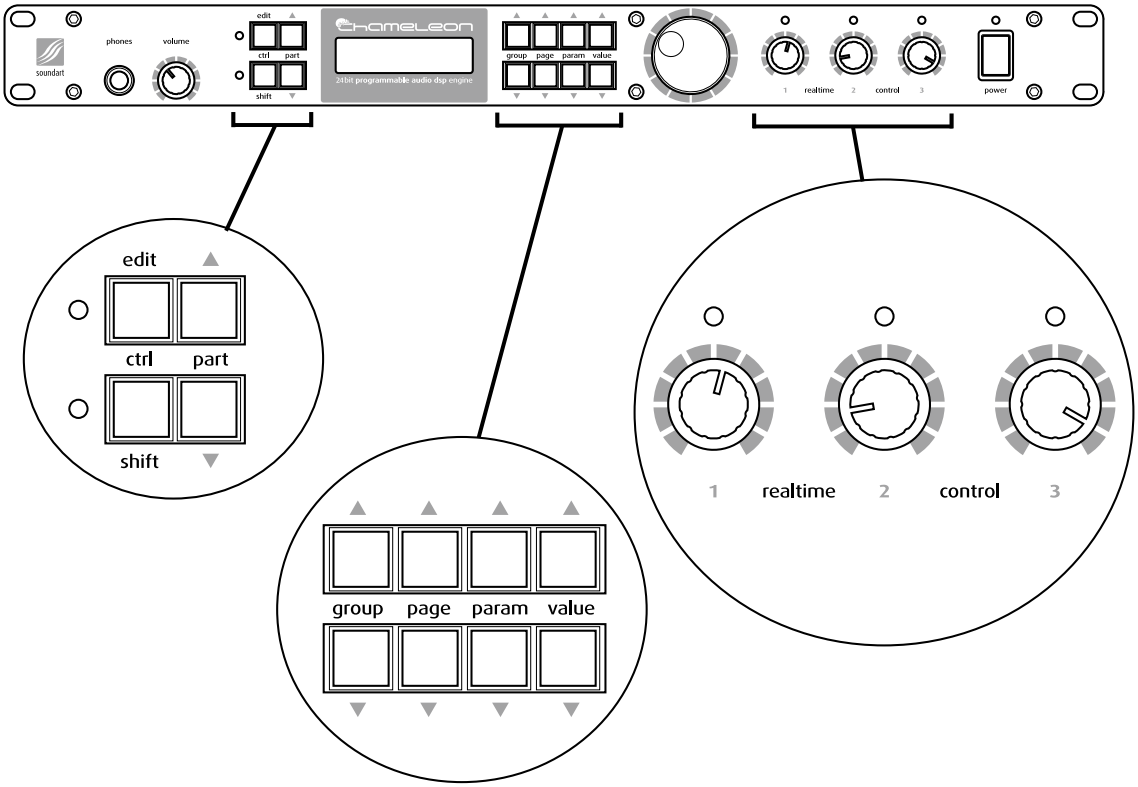
- A backlight LCD display with 2 rows of 16 alphanumeric characters each. It has a predefined ASCII character set. Each character is generated on a 8 rows by 5 columns dot matrix. Additionally, eight custom characters can be defined by specifying the 8x5 "bitmap" for each one.

- Five LED diodes. Three of them are placed over each of the realtime potentiometers, and the remaining two are placed beside the "edit" and "shift" keys, although its functionality is not linked with these controls in any way, and once again it is the programmer task to assing what they indicate when turned on, off, blink, etc.

- A stereo TRS ¼" jack headphones output which is bridged with the device audio line ouput.

Figure 2.2

Front Panel of the Chameleon



From the programmer's point of view, access to all of these controls is performed through the ColdFire. Therefore a library is provided implementing all the necessary functions to access these controls (reading the state of each potentiometer, the encoder and the push buttons, switching on/off the LED diodes and displaying information on the LCD). These functions will be described on Chapter 3, dedicated to the Software aspects of the Chameleon.

2.5 Rear Panel

Figure 2.3 shows the rear panel of the Chameleon, where all the device's connections are placed (except the headphones connector, which is placed on the front panel). These connections are, from left to right:

- 9 VDC/1.2A Power supply connection.
- A MIDI input and ouput, with standard DIN 5 connectors.
- Two analog non balanced input channels, with TRS ¼" jack connectors.
- Two analog non balanced output channels, with TRS ¼" jack connectors.
- A DB-9 connector (RS-232 standard) for debugging functions.

Figure 2.3

Rear Panel of the Chameleon

Chapter

3 Software Architecture

3.1 Introduction

The Chameleon's distinctive characteristic lies in the fact that it is a fully programmable system. To allow the programmer full access to all the physical resources, a software structure (including all the necessary tools) has been designed to allow access in an easy and transparent way. This chapter describes all the software aspects that the developer has to keep in mind when programming the Chameleon.

Programs generated by the Chameleon user (applications) can be downloaded on any other Chameleon by using a standard MIDI sequencer or via the RS-232 connection. Such applications will usually be generated by using the Chameleon Development Environment, CDE (although other development environments can be used, such as Microsoft's Visual Studio). A second tool, the Chameleon Toolkit, will be used to debug the applications running on the device itself and finally to generate distributable files (either MIDI or propietary for RS-232 formats), ready to be downloaded on any other Chameleon.

All the provided development tools work only on PC with Windows 98 or higher operating system (Windows 98/ME/NT/2000/XP). Any other Windows version could work but it hasn't been tested and therefore proper operation is not guaranteed.

Normally, during the debugging process applications will be downloaded and executed on the ColdFire DRAM volatile memory. Once the applications are debugged, these can be stored on the non volatile and rewritable (FLASH) memory, so that they will be automatically executed any time the system starts and is initialised. The portion of the FLASH memory not utilized by the application can be used by the application itself to store the data that user wishes to maintain once the device is turned off (such as presets, configuration, sounds, etc.).

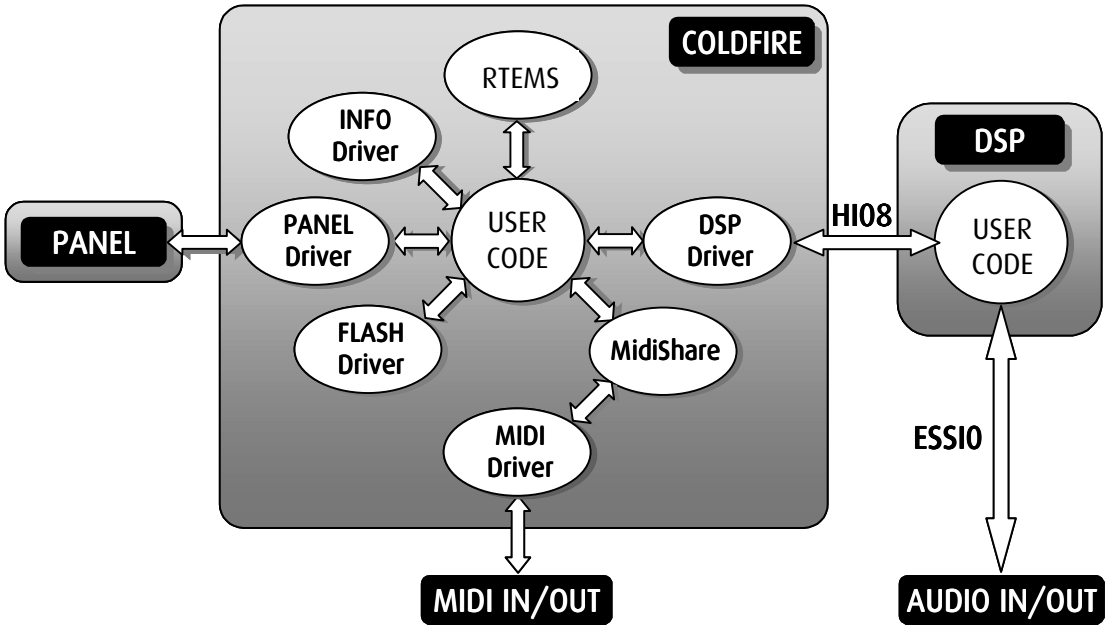
The method applied to generate and store the applications into the Chameleon internal memory uses a highly secure confidentiality and protection scheme, based on the device's serial number. This feature allows the developers to sell their applications with the guarantee these applications will work only on the device with the serial number for which they were generated, and not on any other device. The programmer is free to decide if he wants to generate his application to

run on any device to be freely distributed or to run on a single device to avoid piracy.

Figure 3.1 shows all the available functional software blocks and how these interact one each other.

Figure 3.1

Chameleon's Software Architecture.



The user's code comprises the application itself, which is formed by a control part and a processing part. Henceforth, all references to "the application" will be made without distinction to the control or the processing part, or both, depending on the context (Mainly ColdFire or DSP).

The whole control part runs on the ColdFire. This code can be written in ColdFire assembler (not recommended due to possibly unnecessary complexity) or in C/C++. This code is then compiled and linked with the appropiate tools. Drivers are available to access all the physical system resources (panel, DSP, MIDI, FLASH memory and information), as well as the RTEMS operating system's own services (tasks, timers, message queues, etc.).

The processing code part runs on the DSP. Usually that code will be written in the DSP assembly language (Optionally the code can be written in C, but is not recommended as efficiency will decrease). This code is exclusively specialized in handling audio signals, and in less degree in the communication with the Coldfire-side application part.

The device has a boot program (firmware) which executes all the necessary configuration and initialization tasks, and automatically transfers the whole system control to the application that is stored in FLASH memory (or it remains in waiting state if there is no application stored).

3.2 ColdFire

The whole application is downloaded and executed on the ColdFire, which is the main core for the control tasks. Initially the application part of the DSP is included with the specific ColdFire code. During the initialization process, ColdFire must supply the DSP with its own part of code, by making use of a specific driver. This mechanism is explained in the subsequent sections.

The code for the ColdFire is written in C/C++ language or ColdFire assembler, and is compiled with the processor's specific GNU development tools set. An application is made of a set of user subroutines that make calls to the operating system and to the provided libraries, plus the application's own libraries and the object code of the DSP program converted into data for the ColdFire (see Chapter 4, dedicated to the development tools).

The Chameleon SDK has a built-in set of object code libraries which simplify the task of creating applications and provide the programmer with a standard computer's typical environment, optimized for audio and MIDI data. These libraries include:

- RTEMS, a Multitask Real Time Operating System.
- C Standard library (printf, open, strlen...).
- Math standard floating point (emulated in the ColdFire) library (sin, cos, log...).
- MidiShare, a MIDI Operating System.
- Specific functions library to access and control the Chameleon's resources in a transparent way.

3.2.1 RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is a priority based and event orientated multitasking realtime operating system. It was developed by On-Line Applications Research Corporation (www.oarcorp.com) for a missile guided system for the american governement, and it is currently open and freely distributed, with continued support and updates. RTEMS is available for several hardware platforms, among which is the Motorola 68000 family (from which the ColdFire derives).

RTEMS simplifies the development of complex applications, allowing the programmer to make a functional division into tasks (processes), and providing the necessary directives for synchronization and communication of all the created tasks, which are executed in a multiprocessing environment. The operating system takes control over the Chameleon hardware and offers the programmer an abstraction of it

through a flexible drivers system (with the typical open/close, read/write, ioctl calls), and so controlling and simplifying the different tasks access to the low level resources.

The Chameleon SDK includes RTEMS versión 4.5.0, modified by Soundart to adapt it to the Chameleon's specific demands. From a programmer's point of view, these changes are virtually transparent, except for the following features, which are not available:

- System configuration and initialization.
- Interrupts.
- Multiprocessor support.
- Dual memory access handler.

The remaining high level RTEMS features available for the programmer in the operating system library are:

- Tasks.
- System Clock.
- Timers.
- Semaphores.
- Message queues.
- Events.
- Signals.
- Dynamic memory handling (Regions and Partitions).
- Device drivers.
- Rate monotonic services (Running periodical tasks with extended time precision).
- Extensions (Posibility to process the system's critycal events).

Tasks are the main component in an RTEMS application. There exists the possibility to create multiple tasks which run in a "simultaneous" and asynchronous fashion, each with different priorities, thus allowing an overall functional simplification. It is possible to dynamically create, pause, restart, suspend or delete tasks as a response to given events inside an application. Tasks can make use of semaphores, message queues, events or signals to synchronize and communicate between each other, and timers and rate monotonic to perform periodical actions.

For instance, a typical Chameleon generic application could include a task that periodically checks the front panel for the user input and another

task to monitor incoming MIDI events from the MIDI port. A message queue could be used by these two tasks to communicate such events to a third task. This third task, could process and recognize the incoming events, and then use the DSP driver to send the appropiate command or data to the processing program running on the DSP, which would be appropiately updated.

The RTEMS standard configuration and initialization procedure is automatically performed by the device firmware. Unlike the standard RTEMS version, the Chameleon SDK version only requires specification of three parameters: a function called *rtems_main*, and two variables, *rtems_workspace_start* and *rtems_workspace_size*.

Each Chameleon application must include a function named *rtems_main* (in fact it is actually a task), which is called by RTEMS once it has initialized its services. Usually this main task performs the application initialization, creates and runs the required application specific tasks and allocates the necessary resources. Also two variables must be defined: one called *rtems_workspace_start* and another one called *rtems_workspace_size*. The first one must be an array of *rtems_unsigned32* data (See the RTEMS documentation for the used data types reference), with a size specified by *rtems_workspace_size*. These two variables define a memory space called *workspace* which will be used by RTEMS to house its internal data structures (The maximum RTEMS resources amount, like tasks, semaphores, etc. is not limited, but dynamically assigned by using the available space inside the *workspace*). A typical 128 Kbyte value is enough for most of the applications, although it may be necessary perhaps to increase this value for applications that use many operating system resources.

3.2.1.1 Available RTEMS Functions Listing

Following is a detailed list of the RTEMS functions available for the Chameleon applications programmer, grouped into their functionality inside the system. Those standard RTEMS functions that are not included in the Chameleon specific RTEMS version have been excluded. For a complete reference of these functions and its data types, plus a detailed RTEMS description, please refer to the *RTEMS C User's Guide* provided in the Chameleon SDK documentation.

```
/* TASKS */
rtems_task_create
rtems_task_ident
rtems_task_start
rtems_task_restart
rtems_task_suspend
rtems_task_resume
rtems_task_is_suspended
rtems_task_set_priority
rtems_task_mode
rtems_task_get_note
rtems_task_set_note
rtems_task_wake_after
rtems_task_wake_when
rtems_task_variable_add
rtems_task_variable_get
rtems_task_variable_delete

/* CLOCK */
```

```
rtems_clock_set
rtems_clock_get

/* TIMER */
rtems_timer_create
rtems_timer_ident
rtems_timer_cancel
rtems_timer_delete
rtems_timer_fire_after
rtems_timer_fire_when
rtems_timer_reset

/* SEMAPHORES */
rtems_semaphore_create
rtems_semaphore_ident
rtems_semaphore_delete
rtems_semaphore_obtain
rtems_semaphore_release
rtems_semaphore_flush

/* MESSAGE QUEUES */
rtems_message_queue_create
rtems_message_queue_ident
rtems_message_queue_delete
rtems_message_queue_send
rtems_message_queue_urgent
rtems_message_queue_broadcast
rtems_message_queue_receive
rtems_message_queue_get_number_pending
rtems_message_queue_flush

/* EVENTS */
rtems_event_send
rtems_event_receive

/* SIGNALS */
rtems_signal_catch
rtems_signal_send

/* PARTITIONS */
rtems_partition_create
rtems_partition_ident
rtems_partition_delete
rtems_partition_get_buffer
rtems_partition_return_buffer

/* REGIONS */
rtems_region_create
rtems_region_ident
rtems_region_delete
rtems_region_extend
rtems_region_get_segment
rtems_region_return_segment
rtems_region_get_segment_size

/* DEVICE DRIVERS */
rtems_io_lookup_name
rtems_io_open
rtems_io_close
rtems_io_read
rtems_io_write
rtems_io_control

/* RATE MONOTONIC */
rtems_rate_monotonic_create
rtems_rate_monotonic_ident
rtems_rate_monotonic_cancel
rtems_rate_monotonic_delete
rtems_rate_monotonic_period
rtems_rate_monotonic_get_status

/* EXTENSIONS */
rtems_extension_create
rtems_extension_ident
rtems_extension_delete

/* FATAL ERROR */
```

```
rtems_fatal_error_occurred

/* SUPPORT */
rtems_build_name
rtems_get_class
rtems_get_node
rtems_get_index
```

3.2.2 MidiShare

MidiShare is a MIDI multitasking real time pseudo-operating system, developed by Grame in France (http://www.grame.fr/Midishare). As RTEMS, it is a freely distributed open platform, with periodical updates. The Chameleon SDK includes MidiShare version 1.86.

All the MIDI functionality required by an application for the Chameleon can be implemented by making use of MidiShare. This system implements specific handles to manage MIDI events, such as event memory handling, synchronizing and timing, MIDI tasks and communications.

Communication between MIDI tasks is based on high level events instead of byte packets, which use to be inefficiently addressed by the traditional memory handlers.

Furthermore, MidiShare includes a library called Player, which implements a complete multitrack MIDI sequencer, with track select and syncronizing selectable type (internal, external, SMPTE, MTC, ...), and the possibility to read standard MIDI files.

In the Chameleon architecture, MidiShare uses RTEMS for its low level functions, such as the MIDI port access trough a driver. This way the MIDI communication in an application is totally transparent for the programmer, which has a flexible and robust environment with a highly precise timing.

3.2.2.1 Available MidiShare Functions Listing

The following is a list of the available MidiShare functions. For a complete referenc, please refer to the *MidiShare Developer Documentation* and the Player Reference Manual, both provided with the Chameleon SDK documentation (See documentation index).

```
/*  MIDISHARE ENVIRONMENT */
MidiShare
MidiGetVersion
MidiCountAppls
MidiGetIndAppl
MidiGetNamedAppl
MidiOpen
MidiClose
MidiGetName
MidiSetName
MidiGetInfo
MidiSetInfo
MidiGetFilter
MidiSetFilter
MidiGetRcvAlarm
MidiSetRcvAlarm
```

```
MidiGetApplAlarm
MidiSetApplAlarm
MidiConnect
MidiIsConnected
MidiGetPortState
MidiSetPortState
MidiGetTime
MidiSendIm
MidiSend
MidiSendAt
MidiReadSync
MidiWriteSync
MidiCall
MidiTask
MidiDTask
MidiForgetTask
MidiCountDTasks
MidiFlushDTasks
MidiExec1DTask

/* EVENTS */
MidiFreeSpace
MidiNewCell
MidiFreeCell
MidiTotalSpace
MidiGrowSpace
MidiNewEv
MidiCopyEv
MidiFreeEv
MidiSetField
MidiGetField
MidiAddField
MidiCountFields
MidiCountEvs
MidiGetEv
MidiAvailEv
MidiFlushEvs

/* SEQUENCES */
MidiNewSeq
MidiAddSeq
MidiFreeSeq
MidiClearSeq
MidiApplySeq

/* FILTERS */
MidiNewFilter
MidiFreeFilter
MidiAcceptPort
MidiAcceptChan
MidiAcceptType
MidiIsAcceptedPort
MidiIsAcceptedChan
MidiIsAcceptedType

/* STREAMS */
MidiStreamInitMthTbl
MidiStreamInit
MidiStreamReset
MidiStreamPutEvent
MidiStreamGetByte
MidiStreamCountByte
MidiParseError
MidiParseInit
MidiParseReset
MidiParseInitMthTbl
MidiParseInitTypeTbl
MidiParseByte

/* PLAYER */
OpenPlayer
ClosePlayer
StartPlayer
ContPlayer
StopPlayer
PausePlayer
SetRecordModePlayer
```

```
RecordPlayer
SetRecordFilterPlayer
SetPosBBUPlayer
SetPosMsPlayer
SetLoopPlayer
SetLoopStartBBUPlayer
SetLoopStartMsPlayer
SetLoopEndMsPlayer
SetSynchroInPlayer
SetSynchroOutPlayer
SetSMPTEOffsetPlayer
SetTempoPlayer
GetStatePlayer
GetEndScorePlayer
ForwardStepPlayer
BackwardStepPlayer
GetAllTrackPlayer
GetTrackPlayer
SetTrackPlayer
SetAllTrackPlayer
SetParamPlayer
GetParamPlayer
InsertAllTrackPlayer
InsertTrackPlayer
MidiFileSave
MidiFileLoad
MidiFileLoadFromMemory

/* DRIVER MANAGEMENT */
MidiRegisterDriver
MidiUnregisterDriver
MidiCountDrivers
MidiGetIndDriver
MidiGetDriverInfos

/* SLOT MANAGEMENT */
MidiAddSlot
MidiGetIndSlot
MidiRemoveSlot
MidiSetSlotName
MidiGetSlotInfos
MidiConnectSlot
MidiIsSlotConnected

/* SMPTE SYNCHRONIZATION */
MidiGetSyncInfo
MidiSetSyncMode
MidiGetExtTime
MidiInt2ExtTime
MidiExt2IntTime
MidiTime2Smpte
MidiSmpte2Time
```

The functions listed for streams management form an additional library called MidiStream which is undocumented in the MidiShare distributed documentation. Basically, the MidiStream library is used to convert a typical stream of MIDI bytes into MidiShare MIDI events and back. The function prototypes for this library are in the files:

Chameleon.sdk/include/midishare/EventToMidiStream.h

and

Chameleon.sdk/include/midishare/MidiStreamToEvent.h.

The source code of the functions can also be found in the folder:

Chameleon.sdk/src/lib/midishare/

These functions use an internal table to make the conversion and therefore before using them it is necessary to initialize the tables by calling MidiStreamInitMthTbl and MidiParseInitTypeTbl. The conversion status itself is stored in the datatypes Ev2StreamRec and StreamFifo, which need to be initialized the first time with the appropiate table.

The functions MidiFileLoad and MidiFileSave also have to be commented. Since no file system is supported currently, both functions will return allways error. The function MidiFileLoadFromMemory has to be used to simulate MIDI files in memory.

3.2.3 Chameleon Specific Libraries

The access and control of all the Chameleon resources is performed through the use of specific drivers for each of them, thanks to the facility provided by RTEMS for the hardware access by using *ioctl* calls.

The Chameleon resources that can be accessed by the programmer by means the use of driver are the following:

- Front Panel.
- DSP.
- FLASH Memory.
- Information.

To make the programmer's work easier, a set of high level routines is provided to access these drivers. The using mode of such routines is the same for each resource. Before using a resource, its driver has to be previously initialized through the call to a *init* function, which yields a handle variable (*handler*). This handler is used for the subsequent calls to the rest of the resource specific functions while it is used. When finally this resource is not needed, the handle is freed with a call to a *exit* function.

The definition of all those functions is in the header file *chameleon.h* together with a set of useful macros. Following subsections describe the functionality of each Chameleon drivers. For a complete reference of all the functions in this library, please refer to the *Chameleon API Programmer's Reference* en the documentation.

3.2.3.1 Panel Driver

The front panel driver brings to the programmer a set of high level functions to access the elements that integrate the Chameleon front panel. By making use of these functions, the application routes the input of user inputs and shows him information about its current state. Thus, these routines allow to wait and process panel events, such as button pressings or potentiometer or encoder movements, writing text and redefining characters on the LCD display, and turn on/of the LEDs.

The following are the available front panel driver functions:

```
panel_init
panel_exit
panel_out_lcd_clear
panel_out_lcd_print
panel_out_lcd_redefine
panel_out_led
panel_in_new_event
panel_in_potentiometer
panel_in_keypad
panel_in_encoder
```

3.2.3.2 DSP Driver

The DSP driver gives the access to all the functionality present in the connection bus between ColdFire and the DSP (the DSP HI08 interface). Initially the application downloads the previously compiled signal processing code to the DSP, by making use of the *dsp_init* function. Once this code is running on the DSP, the ColdFire application can exchange data with it (commands, parameters, coefficients, etc.) by using the provided functions.

Calls to the DSP driver can block the calling task in a cooperative way (other tasks continue running), since the communication with the DSP uses interrupt and it is automatically managed by RTEMS.

The system is prepared for future Chameleon models which could incorporate more than one DSP. In fact, to obtain a handle for the DSP driver requires to specify a DSP number, which in the current version is allways 1.

ColdFire can send and receive data, to send commands and to read and write flags from/to the DSP, and the DSP can send and receive data and write and read flags to/from the ColdFire. These possibilities are further detailed on Section 3.3.1 dedicated to the DSP HI08 port.

The available DSP driver functions are:

```
dsp_init
dsp_exit
dsp_read_data
dsp_write_data
dsp_write_command
dsp_write_flag0
dsp_write_flag1
dsp_read_flag2
dsp_read_flag3
```

It's important to note that the data transfers between the ColdFire and the DSP are of 24 bit wide words. Nevertheless, since there is not a data type available with such size, the DSP driver functions to read and write data (*dsp_read_data* and *dsp_write_data*) use the data type *rtems_signed32* (32 bit signed words) for these transfers, with the useful bits right aligned (the 24 LSB), and the most significant byte without a valid content.

3.2.3.3 FLASH Memory Driver

Applications can use the FLASH memory to store data that must be permanently stored, even if the device is turned off, such as configuration data, user presets, sound banks, patches, etc. The FLASH memory driver simplifies the access to this memory type and manages it in a similar way to the DRAM memory.

If an appliction is running in debug mode (i.e. it is running directly from the ColdFire DRAM memory), it won't be able to actually write data into the FLASH memory. Writing will be simulated by using DRAM memory instead. This is due to the fact that it is possible that the FLASH memory had stored a different application (or a different version of the same application that is being debugging), and it would be possible to overwrite part of such application so it could result unusable or with important user data lost. For that reason, writing in FLASH memory by applications running in DRAM memory is disabled, and these only will effectively write when running from there[2].

It is posible to query the FLASH memory size available to user data, which depends on the size used by the application. It's not possible to write to this memory beyond these limits, and therefore the application cannot overwrite itself when it is stored in FLASH.

Following are the available functions to manage the FLASH memory:

```
flash_init
flash_exit
flash_read_data
flash_write_data
flash_get_size
```

3.2.3.4 Information Driver

Information driver is provided to allow the programmer to consult device's own data such as the serial number and the Chameleon model (in prevision to future models).

The device model identifier allows the application to know the available resources (Such as available controls on the front panel, number of DSPs, etc. Which can be model-specific) among different models (currently the Chameleon model available is only #01).

The device serial model is merely informative. The protection mechanism available to the programmers to prevent not allowable application copies to be executed on certain devices is described in Section 4.5 dedicated to the Chameleon Toolkit.

Available functions for the Information driver are:

[2] Please note that this only affects an application while it is being debugged, and not when it is finished and ready to be used.

```
info_get_serial_number
info_get_model
```

3.2.3.5 Macros and utility functions

The Chameleon library includes a set of auxiliary macros and functions specially useful to the programmer.

For the debugging tasks the TRACE and ASSERT macros are available, which allow to show messages on the Toolkit's terminal that informs of specific variables value, error conditions and so on. Calls to these macros will be compiled only in debug mode. When compiling in release mode these calls are ignored.

Since ColdFire works with 32 bit words and floating point fractional arithmetic and the DSP uses 24 bit words fixed point, the appropiate conversion functions between both data types are supplied. These functions are *fix_to_float*, *float_to_fix* and *float_to_fix_round*.

Complete reference for these macros and functions can be found on the *Chameleon API Programmer's Reference* (See documentation index).

3.2.4 Firmware

Chameleon uses a boot program stored in ROM that initializes all the system's hardware components and guarantees the device operation even when faulty applications are stored in FLASH.

Thanks to this boot routine, user aplications are unconcerned from any system initialization task, and only have to take care of its own initialization and execution. DSP is also initialized at boot time, and it remains ready for the application to download the specific code on it. Thus the DSP has the whole system initialized, and it only has to take care of enabling the services that it's going to use, and perhaps to initialize some register with specific values to its demands: audio transmission and/or reception on the ESSI0 port, interrupts, DMA and communication with the ColdFire through the Host port.

If during the booting process some harware or software failure is detected on the Chameleon, system will notice it to the user with appropiate error messages on the LCD display. In case of malfunction on the communication with the panel or on the LCD, the error will be indicated with a 1 KHz tone on the audio outputs.

The booting process is very fast, and if there's some application stored in FLASH, it will be executed in a transparent way.

During the execution of an application on the Chameleon, it is possible to download a new one by using the Toolkit application through the RS-232 debugging port, either to be executed in DRAM (debug mode) or to be

stored and executed in FLASH. This will be a frequent operation on a typical application debugging process. To avoid intermediate filtering on the MIDI reception, it's not possible to perform that operation by making use of the MIDI port. To be able to download a new application via the MIDI, using a standard MIDI sequencer, it is necessary that the device is in waiting mode.

It is possible to boot the system in waiting mode (without loading the applocation stored in FLASH) by turning on the device while maintaining the SHIFT key pressed. The device will display the message "Chameleon #01 (WAITING)" on the LCD. In this state, system allows writing the FLASH memory or running an application in the DRAM memory with the information received either on the MIDI port or on the RS-232 interface. Furthermore, the user can check the device serial number (by pressing the SHIFT key), and the boot subsystems version (by pressing the EDIT key).

It has to be kept in mind that if a buggy or corrupted application is stored in FLASH, it can cause the device not to respond after the boot, as applications keep the overall system control when executed. If this ever happens, the Chameleon boot program will not report any error (since the system's hardware and software are OK), but once the loaded application performs some incorrect action, the device will not operate properly or simply it will stop operating. When this happens, it is allways possible to reboot in waiting mode and then store a correct application.

3.3 DSP

When an application starts, the DSP has not any program running on it, although its memory and peripherals have been previously initialized by the boot program, so they can be directly used by the user code without additional programming needed. The application running in the ColdFire is the responsible for the DSP code downloading when it's initializing. Once the DSP code is downloaded, the DSP driver on the ColdFire takes care to make it execute, so for the programmer "to download the DSP code" is equivalent to "execute it".

DSP programming can be done without any restriction of any kind in terms of registers configuration or memory and peripheral access. However, it has to be emphasized that access to the peripheral and configuration registers should not be done[3], except the ones related to the high speed asynchronous serial ESSI0 interface (enable/disable transmission and/or reception, DMA or interrupt configuration, or poll its status bits) , which acts as the link with the AD/DA converter, and the ones related to the HI08 Host port (configure it for interrupts or DMA, accept ColdFire commands, poll status bits and read or write flags). Any other DSP register different than stated on the following sections and the

[3] Any other attempt to modify the DSP configuration data apart from these stated is **completely advised against**, and in any case will contribute to improve the device's operation.

general purpose registers (X, Y, A, B, Rx, Mx, Nx), simply can be "ignored" with the object of Chameleon programming.

3.3.1 HI08 Host Port

The communications between the DSP and the ColdFire are performed through the DSP HI08 Host interface. It is an 8 bit wide full-duplex parallel port. On the Chameleon, the transferred data types will be mostly control data.

As it was explained in the previous chapter, communication between the DSP and the ColdFire on the ColdFire side is handled by making use of the specific DSP driver. Thus from the ColdFire point of view, to communicate with the DSP results somehow transparent by making use of the available high level function calls.

On the other end, the DSP, handling of such data transfers is slightly more complex, as the programming is done at low level. Programmer has to determine the transfer data type, among the possible ones, that its application is going to need and perform, and so enable the necessary Host port resources. Anyway, since the configuration and the involved protocol signals are preconfigured beforehand, the resulting HI08 port programming model is highly simplified for the Chameleon programmer, and it affects exclusively to the way he wants its application to manage the data. For him, the Host port simply "already works".

The main DSP HI08 port registers that programmers have to kep in mind are:

- HCR: Host Control Register
- HSR: Host Status Register
- HTX: Host Transmit Register
- HRX: Host Receive Register

The possible data transfer types form the ColdFire to the DSP are basically data, commands and flags:

- **Data**: ColdFire sends a given Word number to the DSP, by calling the *dsp_write_data* function. The meaning of such data is completely application dependent. For instance, it can be coefficient tables, parameter changes due to a certain MIDI or front panel event, sound banks, etc. These data are received by DSP on his HRX Host port register. DSP can read it by polling, by enabling host data reception the interrupt, or by making use of a free Direct Access Memory (DMA) channel. These three possibilities are examined further on subsequent paragraphs.
- **Commands:** By writing commands on the DSP Host Port, ColdFire can force the execution of any of the 128 possible interrupt handling routines on the DSP, without having occurred the interrupt signal

itself. Most frequent utility of such feature resides in the fact that the DSP has reserved interrupt vectors (between the program addresses $000064 and $0000FE) for application specific routines. It is possible to make the DSP to execute particular event responses this way.

When the ColdFire writes a command to the Host port (calling the *dsp_write_command*), it writes the interrupt vector address (divided by two) that wants to execute. For these interrupts to be actually executed, DSP has to enable them previously by setting the HCR_HCIE bit[4] (Host Command Interrupt Enable) on the HCR register.

For instance, a particular command could indicate to the DSP that data previously previously written on the HRX register corresponds to the audio output master volume. The interrupt handling routine will read this value and will properly update the application.

- **Flags**: The Host status register HSR has two general purpose bits (HSR_HF0 and HSR_HF1) which can be written by the ColdFire by calling the function *dsp_write_flag*. The meaning of such bits is application dependent and can be used by the ColdFire to signal specific information to the DSP.

The data transfers from the DSP to the ColdFire are similar, but commands can not be used, as explained below:

- **Data**: DSP cand send Words to the ColdFire by writing it on the Host port HTX register. The ColdFire application has to read periodically (depending on the expected reception frequency) these data by calling the *dsp_read_data* function.

- **Flags**: The host control register HSR has two general purpose bits (HCR_HF2 and HCR_HF3) which can be written by the DSP to signal particular application dependent information the the ColdFire. These bits can be read by the ColdFire by calling the *dsp_read_flag* function.

The data transfers in both directions can be done through on of the following possible mechanisms:

- **Polling**: The DSP program periodically consults the host status register HSR which contain information about the transmission status. These bits are HSR_HRDF (Host Received Data Full), which informs that a valid data is received on the HRX register, and HSR_HTDE (Host Transmit Data Empty), which informs that the HTX host transmission register is empty and a new data can be written to be transmitted. This is the simplest method, but also the less efficient. The host port flags 0 and 1 only can be chacked by polling.

- **Interrupts**: An interrupt is available for each event on the Host port. By enabling the HCR_HRIE bit (Host Receive Interrupt Enable) on the Host control register, an interrupt will occur each time a valid data is

[4] For a complete reference of the used nomenclature for the DSP constants, please refer to the file dsp_equ.asm which is placed on the /Chameleon.sdk/include/dsp directory.

received from the ColdFire on the HRX register. The vector for this interrupt is at address P:$000060.

By enabling the HCR_HTIE (Host Transmit Interrupt Enable) on the HCR register, an interrupt will occur each time a data written on the HTX register has been effectively transmitted to the ColdFire, so the HTX register is empty and a new data to be transmitted can be written on it. This interrupt vector is at address P:$000062.

Finally, by enabling the HCR_HCIE bit (Host Command Interrupt Enable) on the HCR register, an interrupt will be triggered each time a command from the ColdFire has been written. Interrupt vector then depends on the command written by the ColdFire.

The DSP peripheral associated interrupts have an assignable priority level. On the Chameleon, the HI08 Host Port interrupt[5] has a default binary value of IPL = %01.

As any other interrupt on the DSP, for the interrupt be actually triggered and its handling routine be executed, it has to be masked by using the MR_I[0-1] bits on the MR register (MR_I = %11 means to enable all the interrupts).

- **DMA:** A Direct Memory Access (DMA) can be used to automatically transfer the received or transmitted data from/to previously specified memory buffers, without the DSP core intervention. Properly used, this technique is the most efficient as it allows data transfers in parallel to the normal execution of the code (See Chapter 10 on the DSP56300 Family Manual and the Motorola Application Report APR/23, *Using the DSP56300 Direct Memory Access Controller* on the Chameleon SDK documentation).

A detailed description of the HI08 Port can be found in Chapter 6 of the DSP56303 User's Manual.

3.3.2 ESSI0 Port

The DSP56303 owns two full-duplex, high speed synchronous serial communication ports, named ESSI0 and ESSI1. Only ESSI0 is used on the current Chameleon model.

This port is used on the Chameleon as digital audio input and output, and it's directly connected to the AD/DA converter module. It consists of independent transmitter and receiver, which transfer serial data frames synchronously with the AD/DA converter. The converter clock guarantees the timing precission for each transferred audio sample. The ESSI0 port programming model from the Chameleon programmer's point of view is also highly simplified, since all the configuration needed to correctly communicate with the AD/DA converter is previously done and it must

[5] To study the DSP interrupt mechanism in deeper detail, please refer to Section 4.4 in the DSP56303 User's Manual provided in the Chameleon SDK Documentation.

remain unchanged. The only programmer's concern about that is to enable/disable the input and the output, to synchronize left and right channels, and finally to define and configure the data transfer mode for his application (Polling, interrupts or DMA, in a similar way as it is done on the Host port).

ESSI0 port includes an input serial line and three outputs. From these three output lines, only the first (TX00) is used on the Chameleon model #01.

The ESSI0 port registers to have in mind are :

- SSISR0: ESSI0 Status Register.
- CRB0: ESSI0 Control Register.
- TX00: ESSI0 Transmit Data Register 0
- RX0: ESSI0 Receive Data Register

Both audio input and output are stereo. The two input channels (L and R) are recived by the ESSI0 on the same receiver module register (RX0) from the AD converter, and the two output channels are transferred to the DA converter through the same transmitter module register (TX00). The data format in both cases is interleaved samples. Internal format of each sample received and transmitted on the RX0 and TX00 registers is fractional fixed point arithmetic[6], so the DSP Arithmetic Logic Unit can operate with it.

Depending on the algorithm, it may be necessary to enable the input, the output or both, any combination is possible. Output is enabled/disabled by simply writing the CRB0_RE (Receive Enable) bit on the CRB0 register, and the output by writing the CRB0_TE0 bit (Transmit Enable).

As the samples of both L and R channels are alternately received on the same serial port, it is necessary to know to which channel belongs the currently received sample, and the same for the transmitted samples. For that the AD/DA generates a synchrony signal to indicate which channel is currently transmitting. This signal is readed by the DSP on the SSISR0_RFS (Receive Frame Sync) bit. Usually when the DSP code starts, it will wait to read this signal to be synchronized and to know from which channel it's reading. A high value on this bit means that the channel being received is the left, and a low value means the right channel.

The input samples reading is done on the SSISR0_RX0 register and the output samples writing on the SSISR_TX00. The mechanism used by the DSP to read and write the samples can be one of the following:

- **Polling**: The application main loop takes care to poll the SSISR register to know if it has received a new input sample by querying the

[6] For those readers not familiarized with the fixed point fractional arithmetic, it is recommended to read the Motorola Application Report APR3 that can be found on the Chameleon SDK documentation.

SSISR_RDF bit (Receive Data Full), or if the transmission register is empty and a new processed output sample can be written, by querying the SSISR_TDE bit (Transmit Data Empty). This mechanism implies to perform processing sample by sample, and probably it is the most time consuming option.

- **Interrupts**: The ESSI0 port can be configured to trigger an interrupt when a new sample is received on the SSISR0_RX0 or when the SSISR0_TX00 register content has been transmitted and it is empty and available to write the next output sample.

  To enable the reception interrupt, the CRB0_RIE bit (Receive Interrupt Enable) must be enabled, and to enable the transmission interrupt, the CRB0_TIE bit (Transmission Interrupt Enable) has to be set.

  It is posible to detect reception and transmisión errors by means of the reception and transmission with exception interrupts. This interrupts are triggered when the RX0 register has been readed whithout having read the previous sample (overrun) and when the TX00 register hasn't been written on the required interval time (underrun) respectively. These interrupts are enabled by setting the CRB0_REIE bit (Receive Exception Interrupt Enable) and the CRB0_TEIE (Transmit Exception Interrupt Enable) on the CRB0 register.

  The ESSI0 Interrupt priority level on the Chameleon is preassigned to IPL = %11.

  As any other interrupt on the DSP, the ESSI0 interrupts have to be masked by using the MR_I[0-1] bits on the MR register to be actually triggered.

  The interrupt processing mechanism is usually more efficient than pollin, as the audio input and ouput is handled asynchronously to the processing tasks, so the time dedicated to this handling is reduced.

- It is posible to enable a DMA channel to perform the transfers between the input and ouput registers and previously specified memory buffers in parallel to the processing program execution. This way the data transfers does not overload the application. This mechanism of data transfers can be highly efficient for block processing.

Complete reference for the ESSI0 port can be found at the Chapter 7 of the DSP56303 User's Manual.

3.3.3 External Memory

In addition to the 8 KWord SRAM internal memory, DSP has 4 MWord of DRAM EDO type external memory mapped on the address range $400000 - $7FFFFF, completely available to the user application. This memory is shared between all the DSP memory spaces, and so the same address will be accessed independently of X, Y or P memory is being accessed.

DRAM memory is structured in 1 KWord pages. Accessing to an address inside the same page as the previously accessed address (inpage access) requires one waiting state, whereas accessing to an addres in another page (offpage) access requires 8 waiting states.

It is important to keep in mind 7this information when planning the application memory map in optimization terms. Frequently accessed code and data should be placed on internal memory, and less frequently accessed code and data should be placed on the external DRAM memory, trying to group related blocks inside the same page.

Chapter

4 Development Tools

4.1 Introduction

All the necessary tools to generate, debug and distribute Chameleon executable applications are provided in the Chameleon SDK, thogether with quite abundant related documentation. These tools include specific applications developed by Soundart and third party utilities. All of them are freely distributable.

The tools that integrate the whole Chameleon SDK:

- Motorola Suite56™ DSP Tools
- GNU Cross-Platform Compiler Collection
- Chameleon Development Environtment CDE
- Chameleon Toolkit
- Scilab

The Chameleon SDK can be obtained from www.soundart-hot.com or in the CD-ROM supplied with the Chameleon hardware. The latest version of the SDK is allways on the website. If you have bought the hardware Chameleon then you should check to see if there have been any updates to the content.

Next sections explain the SDK components in detail.

4.2 Motorola Suite56™ DSP Tools

The DSP code is compiled and linked by means of the Motorola tools for the DSP563XX (Compiler/Linker/Assembler/Librarian). Another utility developed by Soundart, *cld2header.exe*, converts the generated DSP executable file (a file with extension **.cld*) to a C header file which contains a byte array with the binary image of such executable code, so it can be included with the C/C++ ColdFire code files to be directly downloaded to the DSP by using the *dsp_init* function. As it has been stated, once the code is downloaded it begins to execute at once.

DSP programs are mostly written in the DSP56XXX assembler, and preprocessor directives can be used for compilation. The complete DSP assembler instruction set reference and guide can be found on Chapters 12 and 13 of the DSP56000 Family Manual, and in the Motorola DSP Assembler Reference Manual. There's also available the Motorola DSP Linker/Librarian Reference Manual and the Suite56™ DSP Tools User's Manual. To write DSP programs in C, the GNU DSP563CCC Optimizing C compiler is supplied, together its own User' Manual. For less experienced readers on the Motorola DSPs programming, an introductory tutorial with 8 exercices is available. All that material is completely available on the SDK documentation.

4.2.1 DSP GUI56300 Simulator

In the DSP code debugging process, a simulator plays a very important role. As the Chameleon hardware is beforehand initialized and guaranteed to work properly by the firmware, hardware debugging is not so important, and what actually counts is to be able to debug the algorithms itself, without having to take care about anything else.

It is available a useful and complete simulator (GUI56300), developed by Motorola for its DSP563XX family, which allows to perform detailed simulations of the developed algorithms to check its proper operation, and to obtain profiling data to detect performance lacks and optimize the code.

The simulation level is total, and it is posible to simulate peripheral events and interrupts. The employed clock cycles on specific code blocks can be checked to analyze the optimization results.

Binary stimulus files can be loaded as the algorithm input, and output files can be generated. Thus, Scilab or another mathematical software can be used to generate the stimulus files (impulses, audio files, test signals, etc) to analyze later the processed results.

A complete reference is found at the Suite56™ DSP Simulator User's Manual.

4.3 GNU Compiler Collection

ColdFire programs are generated by using the GNU Compiler Collection Set (www.gnu.org), which include an optimizing C/C++ compiler, a ColdFire assembler/linker/librarian, and the Make utility for closs-platform compilation. These programs run on Windows and generate ColdFire code.

The compilation system uses the Make utility with standard makefiles. Header makefiles which can be included by the application own

makefiles are supplied, to make the tedious project managing task easier. These files are placed on *Chameleon.sdk/make*[7] folder.

Once the source code is written, the Make utility must be called with the appropiate command line parameters to compile and link the source files to obtain a ColdFire executable application. The Chameleon Development Environment automatically performs this call from its Build menu, but the programmer should know several aspects of the Make utility and the structure of the makefiles that have to be created for his application to be build successfully.

When an application is generated, several of the libraries described on Chapter 3 (RTEMS, MidiShare, Chameleon, etc) and placed on the SDK directory structure are used. Search for these libraries is done through makefiles which are suitably placed in such structure. Usually the source files of the aplication will contain calls to functions of such libraries, apart from other user defined header files. Makefiles supplied with the SDK take care to appropiately redirection the compilation tools when the Make utility is called. However, the application has to include its own makefiles, defining certain variables that will instruct the compiler about how to perform certain application-specific actions.

Below the typical application makefiles particularities are explained in detail. It's assumed that these described files include the header makefiles supplied with the SDK. Next is an example about several ColdFire source code files which are on an arbitrary directory, and the DSP souce code files are placed on a subdirectory called "dsp".

The directory which contains the ColdFire source files has to contain a makefile with a structure similar to the following. All the C, C++ and assembler source files in such directory will be compiled, although only those required by the application will be finally linked:

```
APP    := myappname.elf
OUTDIR := examples/myapp

include /Chameleon.sdk/make/main.mak

dsp/dsp_code.h::
         $(call make, ./dsp)
```

The APP variable specifies the ColdFire executable file name to be build, which is dirctly downloadable on the Chameleon (ColdFire executable files have ".elf" extension).

The OUTDIR variable specifies the subpath where the compilation output results will be stored. By default this subpath is added to the */Chameleon.sdk/out/model01/* followed by */debug/* or */release/* directory depending on the compilation mode (debug or release, i.e. by calling Make with the "debug" or "release" command line).

[7] The Chameleon SDK installer takes note of the user spcified installation path, so all the subsecuent references use a relative path as file searching route. All reference to Chameleon.sdk/ refers to the actual SDK installation path. Makefiles use the Unix slash ("/") to specify directory trees.

The rule that refers to the file *mydspcode.h* is used to recompile the source files by the appropiate tool, and so obtain the header file *mydspcode.h* which contains the DSP code binary image. This file will be automatically generated by the *cld2header* program once all the DSP source files placed on the ".\dsp" subfolder are compiled. This rule is executed when some ColdFire source file that requires this header file is found (usually the file that contains the call to the *dsp_init()* function).

Other additional variables which can be specified are:

EXTRAGOALS: points to further targets to be reached.

EXTRAOBJS: specifies additional object code file paths that have to be linked (e.g. compiled user libraries).

CCINCLUDES: specifies header file paths included in some source code files to be compiled and which are in different folders.

These two variables are useful to reuse code such as user created C++ classes, constant definitions, etc.

CCDEFINES: allows to define new symbols that will be defined during the compilation.

Next, the */Chameleon.sdk/make/main.mak* is included, which is one of the supplied makefiles, and which takes the variable values previously defined to perform the compilation and linking of the application. This makefile instructs the compiler and the linker to search for the system libraries, generate dependency lists, etc.

Finally, another makefile has to be created on the subdirectory where the DSP source code resides (in our example ".\dsp"), with a content similar to the following:

```
DSPAPP :=        myappname.cld
OUTDIR :=        examples/myapp/dsp
EXTRAGOALS :=    mydspcode.h

include /Chameleon.sdk/make/maindsp.mak

dsp_code.h : $(DSPAPP)
         $(call cld2header, dspCode)
```

The DSPAPP variable specifies the DSP executable file name to be created. The source file used to obtain this file is the one whose name coincides with its own, and has the ".asm" estension.

The OUTDIR variable specifies the subpath where the resulting output files will be generated, which is added to the default path */Chameleon.sdk/out/model01/* plus */debug/* or */release/* depending on the compilation mode (debug or release). For coherence, on this example this path is a subdirectory of the ColdFire output files one, as it is recommended to proceed usually.

EXTRAGOALS: specifies additional targets to reach. In this case it specifies the name that must have the header file containing the DSP code binary image, which has to be the same to the one specified on the ColdFire makefile previously explained.

The makefile */Chameleon.sdk/make/maindsp.mak* has to be included, which will take the value of the defined variables to direct the compilation of the DSP files.

Finally, a rule is defined to obtain the file *dsp_code.h* properly. It tells *make* to call the *cld2header* utility to generate the header file and allows to specify the name of the array in this file which contains the binary image of DSP code and which is used by the ColdFire function *dsp_init()* to download this code to the DSP (in the example the array will be named *dspCode*).

Additionally to these variables, the following can also be specified:

DSPDEPENDS: specifies dependencies of the DSP source file, so if some of the included files is changed, it will be recompiled again. It must be noted that if these file names are not assigned to this variable, they will NOT be recompiled although they are modified, so these modifications won't be reflected on the final executable.

To obtain more information about the Make utility, please refer to it's complete and extensive reference on the SDK documentation.

4.4 Chameleon Development Environment (CDE)

This utility developed by Soundart consists of a complete integrated development environment which allows to generate and compile code exclusively for the Chameleon. It's thinked as the only tool needed to develop Chameleon applications. It's main features include:

- Project Oriented
- Multidocument graphical interface
- Project compiling process integrated support, by calling the appropiate tools
- Compilation output visualization and direct jump to error and warning messages
- C/C++, Makefiles and DSP assembler language syntax highlighting with bookmarks and multiple undo levels
- Advanced search in and between files with regular expresión support and direct jump to the results
- Use of templates to create files and projects

- Auxiliary development and user tools calls (Chameleon toolkit, DSP simulator, Scilab, Explorer...)

- Highly customizable (colors, tools, menus, skins...)

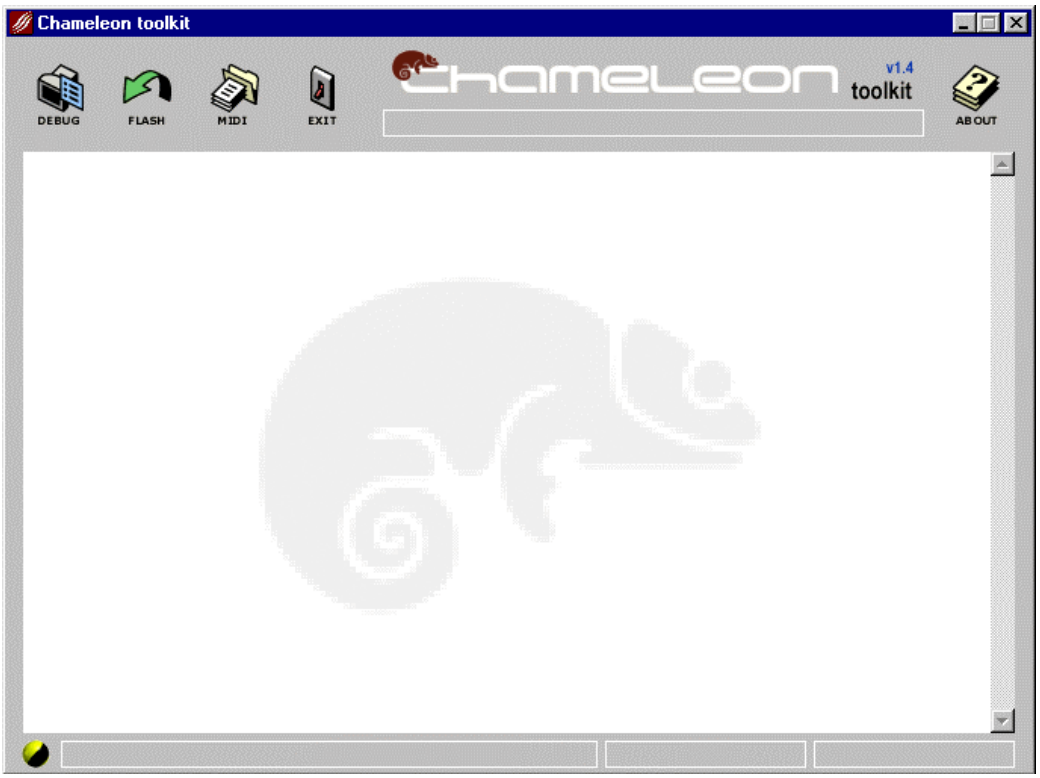
The use of this application is quite simple and straightforward, as it is a Windows application very similar to many of the already existing Integrated Development Environments, with the only particularity that it's exclusively designed to manage Chameleon applications.

4.5 Chameleon Toolkit

Chameleon Toolkit is the tool that allows download and debug the applications. It accesses to the Chameleon through the RS-232 either to download applications that will be executed in the DRAM memory (debug mode) or to store applications in the FLASH memory (release mode). It is also possible to generate standard MIDI files to be used by other users, and to encode them so they only can be executed on a single specific Chameleon. Furthermore, the Toolkit shows a terminal screen during the execution of the applications in DRAM where is possible to display messages sent by the application executed in debug mode, this way allowing to check the variable states, error messages, etc.

The Toolkit window is shown below (**Figure 4.1**). It is quite simple and straightforward. The buttons for Debug, Flash, and MIDI offer various ways to load applications on the Chameleon hardware. The Exit button closes the Toolkit, while the About button displays version information and contact details for Soundart.

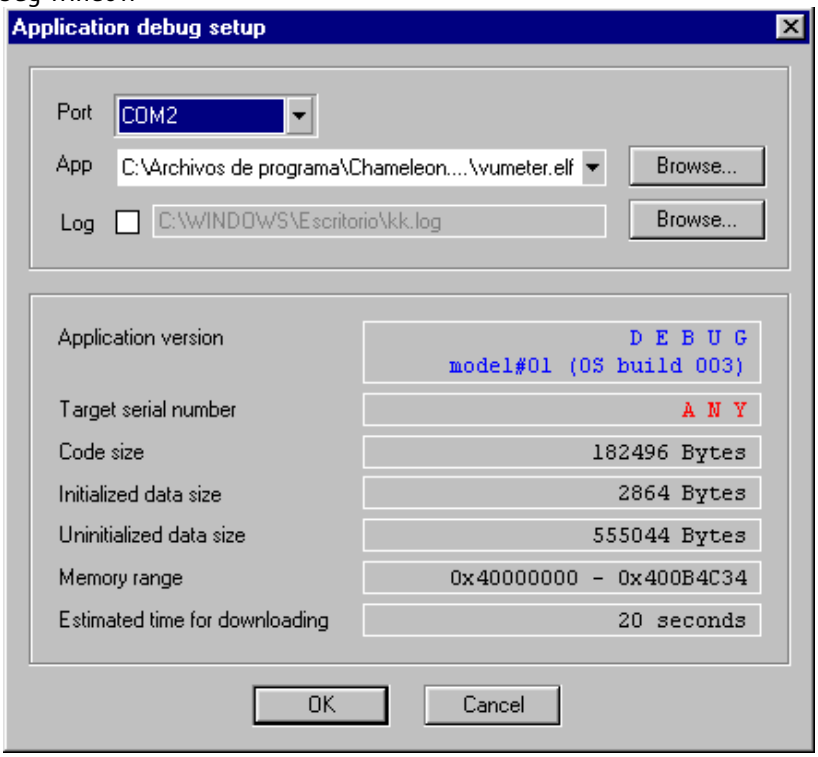
Figure 4.1
Chameleon Toolkit Window.



The main (white) area of the window acts as a terminal. When the user is debugging applications, the Chameleon hardware can send information to this window to report the progress of the software, by using the TRACE and ASSERT macro calls.

Figure 4.2 shows the dialog presented when clicking on the Debug button. It is divided into two panels: the first for the user to enter information, and the second to display information about the selected application. In the upper panel, the first control allows to select the computer serial port to which your Chameleon is connected. By default, this is 'COM 1'. Underneath, user can select an application to load on the Chameleon.

Figure 4.2
The Toolkit Debug window



Underneath, user has the option to record a log file where all the messages received on the terminal from the Chameleon will be stored. Clicking the **Log** checkbox will allow to specify a file to store the details of the transaction. This may be useful when developing a large project or when many messages are sent from the application.

The lower panel is filled with information once an application is selected. The **Target Serial Number** in this dialog will always be 'Any'. In other words, loading an application via the debug interface bypasses any serial number checking on the Chameleon hardware. Later we will see how to add security features to applications.

Below is the **Application Version**. Depending on which mode the loaded application was compiled, this may be either 'Debug' or 'Release'. The debug version of an application contains the extra TRACE or ASSERT macro calls to display information in the Toolkit terminal window while the Chameleon is being used.

Underneath this are some statistics about the application you have loaded. **Code Size** refers to the size of the application being sent to the Chameleon. **Initialised Data Size** refers to variables which are initialised with a starting value in the program. **Uninitialised Data Size** indicates how much memory is allocated for static variables. **Memory Range** always starts above 0x400000.

Finally, an estimate of the time required to download the application to the Chameleon from the computer is displayed.

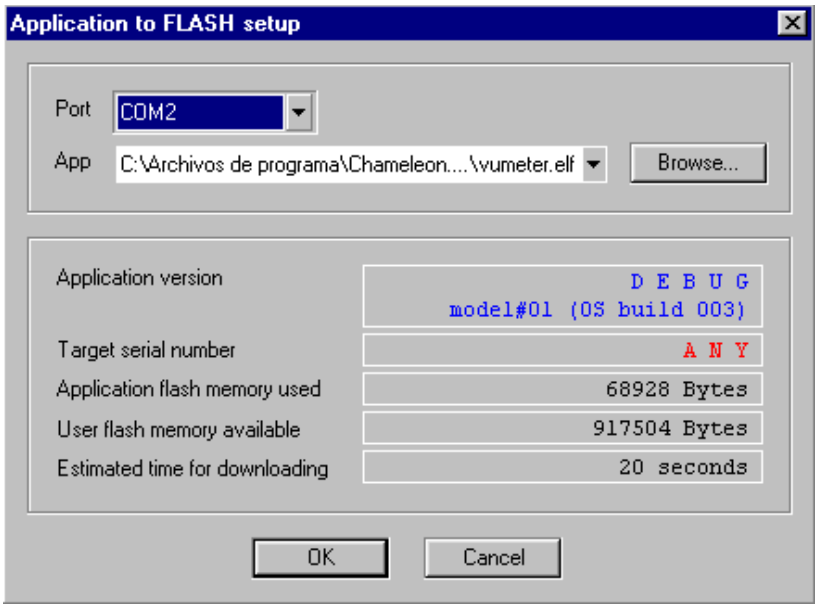
The **OK** and **Cancel** buttons have their normal functions: **OK** will go ahead and attempt to download the application, while **Cancel** will abort the action and return you to the main Toolkit window.

Clicking OK will clear the Debug window and replace it with a download progress indicator. First the Chameleon hardware will be reset, and then data is transferred from the computer to the Chameleon.

Clicking the FLASH button will show the dialog of **Figure 4.3**. It is similar to the Debug dialog, but now it is not possible to log to the terminal, as no debugging is possible when applications run from FLASH. The portion of FLASH memory used and the remaining available to the user are displayed, as well as the estimated downloaded time. Clicking OK will start the download. Once the code is downloaded, the Toolkit will ask the user to confirm the application storage in FLASH. Clicking OK again the application will be effectively stored in FLASH. It will take a few seconds, and the device should not be turned off during the storing process to avoid the data being incompletely stored and have a corrupt application stored.

Figure 4.3

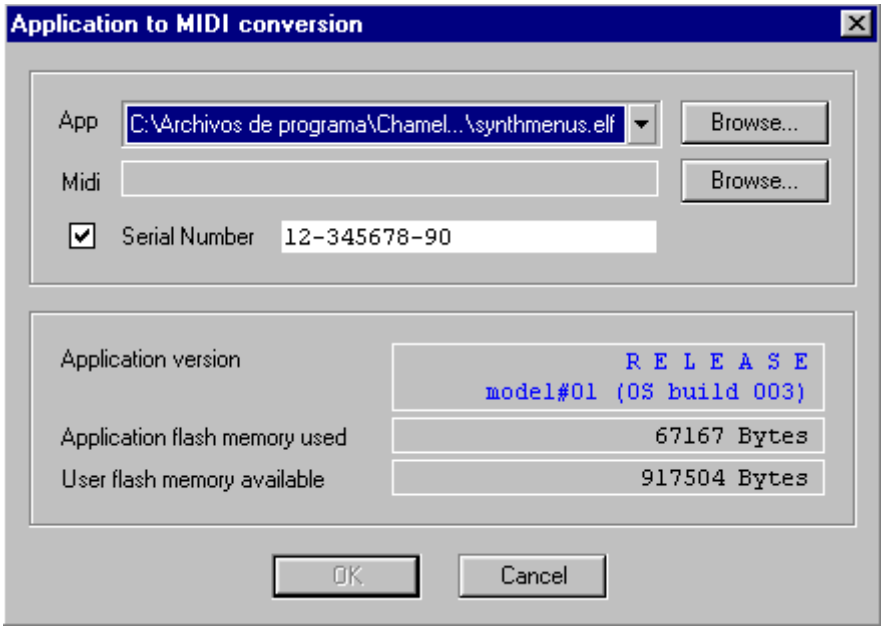
The Toolkit FLASH window



Clicking on the MIDI button will open the dialog showed in **Figure 4.4**. By means of this dialog the user can generate a standard MIDI file containing his application. Any other user will be able to download this MIDI file to the Chameleon by booting it in waiting mode, connecting the Chameleon MIDI input to the MIDI output of any sequencer, and playing this MIDI file. The application will then be stored permanently in the Chameleon.

Figure 4.4

The Toolkit MIDI Window



The application is specified in the "App" labeled box, and the MIDI file to generate in the "Midi" box. By checking the Serial Number checkbox will allow the user to specify for which Chameleon specific serial number he wants to generate the MIDI file. Only valid Chameleon serial numbers are allowed. The generated MIDI file will be then only able to execute on the Chameleon with that serial number, and it won't run on any other machine. The security mechanism is highly robust and the hardware takes care to validate this serial number. Leaving the Serial Number checkbox unchecked will generate a MIDI file executable on any Chameleon whitout regarding its serial number.

4.5.1 Hints on Debugging Aplications

As it has been stated, programmer has available the TRACE and ASSERT macros to generate messages that will be displayed on the Toolkit terminal window. This is currently the only way to debug the executing code. No hardware debug is provided in the current model of the Chameleon, athough it could be provided in future models.

By placing appropiate TRACE and ASSERT messages on the critical parts of the ColdFire code, it is possible to know the states of any variable, what parts of code were not executed under certain conditions, the cause of errors and so on.

For the DSP code debugging, it is highly recommended the use and understanding of the provided simulator, which can be very useful to debug and optimize the applications. The DSP hardware debuggers are mostly intended to solve system hardware problems when developing a new hardware DSP architecture. These problems are not such for a Chameleon programmer, as they are solved beforehand and he has not to take care of it. A software debugger is allways useful even when a

hardware one is available, as there are problems that cannot be otherwise addressed, such as consecutive audio input samples capture, code profiling, etc.

It's also possible to get DSP register values in run time when debugging the applications in the Chameleon, by creating a ColdFire task available only in debug mode. This specific task can read periodically data from the DSP via the host DSP driver. The DSP can write the desired data values in the Host port and the ColdFire task can send the received data to the Toolkit, which will display them on its terminal screen. As usual, it's also possible that the Toolkit logs these incoming messages in a binary data file for further analysis. Once the possible problem is solved, the ColdFire debug can simply be turned off and continuate the normal developing process.

4.6 Accesories

4.6.1 Scilab

With the needs of a Digital Signal Processing applications developer in mind, a complete mathematical software package has been included: Scilab[8]. Developed by the Scilabgroup (INRIA-Rocquencourt, Metalau Project and the Cergrene ENPC, www-rocq.inria.fr/scilab), this freely distribited program allows to perform high level simulations of the algorithms before these are finally implemented in the DSP. It works in a similar way (in several aspects even better) to the existing commercial mathematical and simulation software packages, such as Matlab, Mathcad, etc. The included version is Scilab 2.6, especially recompiled by Soundart to be included in the Chameleon SDK, with some minor bugs fixed.

Introductory material and the complete Scilab Reference Manual can be found in the SDK documentation.

4.7 SDK Structure

The Chameleon SDK installation is divided into four parts to simplify the redistribution:

- **Chameleon Software Development Kit Core Files**: It contains the Chameleon libraries, the system header files, the Soundart utilities for code generation and debugging (Toolkit and CDE), plus sample code.

[8] The Scilab software is not needed to develop applications for the Chameleon, but it is include as an auxiliary tool anyway.

- **Chameleon Binaries**: It contains all the GNU and Motorola compilation tools.
- **Chameleon Documentation**: Complete documentation for the whole Chameleon SDK.
- **Chameleon Silab**: Scilab 2.6 compiled specially for the Chameleon SDK.

It is not possible to install one of these components if it already exists another version (same or different) previously installed. The installation program will warn to the user in such case and it will allow him to uninstall the previous version.

A full installation of the Chameleon SDK comprises over 3,000 files, so a map may be helpful. The following table shows the directory structure created after a complete SDK installation and explain the important items.

Directory	Contents
Chameleon.sdk	Root directory of the Chameleon SDK
├--bin	All binary (executable) files in the SDK are stored here
│ ├-cde	Chameleon Environment Development files
│ ├--coldfire	Coldfire source code compilation tools
│ ├--dsp56303	DSP source code compilation tools
│ └-scilab	Scilab Program Files
│	
├--doc	Documentation for the Chameleon and SDK components
│ ├-chameleon	The Chameleon specific docs
│ ├-coldfire	The Coldfire docs
│ ├-dsp	The DSP56303 docs
│ │ ├-appnotes	Useful Motorola application notes on DSP
│ │ └-other	Useful background and example documents on DSP
│ ├-gnu	The GNU tools docs
│ ├-midishare	MIDIShare system and the Player sequencer docs
│ ├-rtems	RTEMS docs
│ └-scilab	SciLab docs
│	
├--include	Header files containing Chameleon-specific functions
│ ├-Chameleon	Chameleon hardware drivers
│ ├-dsp	DSP56303 include files
│ ├-midishare	MIDIShare include files
│ ├-newlib	newlib, an optimised version of the C Standard Library
│ └-rtems	RTEMS include files
│	
├--lib	Hardware version-specific library files
│ └-model01	Libraries specific to Model #01 of the Chameleon
│ ├-debug	Debug-version system functions
│ └-release	Release-version system functions
│	

Directory	Contents
├--licenses	Legal and licensing information for developers
├--make	Basic makefiles used in all Chameleon applications
├--out	Storage directory for Chameleon executables
│ └-model01	Executables created for Model 1 of the Chameleon
│ ├-debug	Debug versions
│ └-release	Release versions
└-src	Source code for Chameleon applications
└-examples	Example source code included in the SDK

Chapter

5 Where to Go From Here

Since the reader's goal is to keep hands on actual programming, it's logical to think on which will be the next step after or while reading this guide. Following sections explain how to go deeper into the Chameleon step by step. Also, the Appendix A contains the complete SDK documentation index to help you to find the needed information at each moment.

5.1 SDK Code Examples

First thing to do should be to take a closer look at the code examples provided in the Chameleon SDK to get a clear idea about the general Chameleon programming scheme, how files are structured, how the Chameleon resources are handled, how makefiles work and so on. Following a small description about each of the currently available examples.

5.1.1 Hello

We begin with the classic introductory program making our machine display the words 'Hello World'. Although this is a simple example, it demonstrates the basics of interaction with the operating system and the front panel of the Chameleon.

It is found in the folder Chameleon.sdk/src/examples/hello

5.1.2 Welcome

Our second project builds on the first, but is quite a bit more involved. It introduces the Info driver, and allows some interaction with the front panel. We have already written to the display: now we will respond to input. This a simple model for all panel interaction with the Chameleon.

It is found in the folder Chameleon.sdk/src/examples/welcome

5.1.3 Showpanel

In this example, we program the Chameleon to flash the front panel LEDs in a pattern. Building on the last example, adjusting any front panel control will generate a response from the machine.

The ShowPanel application still focuses on the microcontroller, but is considerably more complex. Independent RTEMS processes are demonstrated for the first time, and so is the use of real-time operating functions. We also examine how to redefine characters for the display, and respond to a wider range of front panel events.

It is found in the folder Chameleon.sdk/src/examples/showpanel

5.1.4 Dspthru

With this application we have our first interaction with audio on the Chameleon (this is why you bought it, right?). As the name implies, this program simply reads data from the audio inputs of the Chameleon and echoes it back to the outputs, while allowing you to control the volume. Obviously, to hear it working you'll have to connect something to the inputs and have the volume above zero.

This example shows the fundamentals of handling audio I/O in the Chameleon, in several ways: polling, interrupt and DMA. To select one of these available modes, you have to define one of the symbols POLLING or INTERRUPT to use the polling or interrupt method, or if any of them are not defined, to use the DMA. It only uses the volume potentiometer to adjust the desired volume using a pre-filled table with a variable gain from -90 dB to 0 dB. To smooth the abrupt volume changes, a volume ramping is implemented in the DSP side.

It is found in the folder Chameleon.sdk/src/examples/dspthru

5.1.5 Dspmem

This application shows the basics of interaction with the Chameleon's DSP. The microcontroller boots the DSP, then tests the DSP memory. Considerably simpler than ShowPanel, this example demonstrates the fundamental process used to pass information back and forth between the two processors and how thd DSP accesses its external memory.

It is found in the folder Chameleon.sdk/src/examples/dspmem

5.1.6 Cfthru

This example shows how to get audio data from the DSP to the ColdFire and back again. It only uses the volume potentiometer to adjust the desired volume using a pre-filled table with a variable gain from -90 dB to 0 dB. The volume is updated using a timer procedure to interpolate

intermediate points allowing smoother changes than using directly the valued got from the panel when it is received.

It is found in the folder Chameleon.sdk/src/examples/cfthru

5.1.7 Hostcommands

The way the ColdFire sends data to the DSP using the DSP Host Commands is illustrated in this example. It is based on the dspthru example, with some extended functionality. The volume is now independent for channels right and left, and one pushbutton is used to mute/unmute the audio output. Additionally to the volume potentiometer, the "CONTROL 1" potentiometer is used as a "balance" control, and the "EDIT" key is used as mute/unmute control. Since several commands are sent from the ColdFire to the DSP through the same port (the Host port), two different host commands are used to allow the DSP to interpret correclty the incoming control data stream.

It is found in the folder Chameleon.sdk/src/examples/hostcommands

5.1.8 Midimon

In this example, the main MidiShare features are used to show how incoming MIDI events are handled typically in the Chameleon and sent to the Toolkit to be displayed on its terminal in console mode.

It is found in the folder Chameleon.sdk/src/examples/showpanel

5.1.9 MonoSynth

MonoSynth is a complete monophonic synthesizer application for Soundart's Chameleon with two wavetable oscillators, a white noise generator, a mixer, a resonant lowpass filter with cutoff envelope (ADSR), an amplifier with gain envelope (ADSR) and two stereo delay effect units.

This example covers practically all aspects necessary to implement a typical synthesizer in the Chameleon. Some of the key characteristics are:

- Use of a modular-like DSP processing framework.
- Included DSP code implementing generic blocks of frequently used modules: oscillators, filter, vca, envelopes, delay lines...
- Fast ColdFire to DSP, interrupt based, generic communication framework.
- DSP simulator/profiling support code (using conditional assembly directives).

- Realtime computation of DSP parameters in the ColdFire using standard floating point functions in C.
- Realtime processing of incoming MIDI data, supporting NoteOn, NoteOff, PitchBend, Controllers, ProgramChange and SysEx messages.
- Realtime generation of MIDI controller data using the panel knobs.
- Programmable MIDI-thru-merge engine (MIDI input messages routed to MIDI output in realtime together with the internal generated ones).
- MIDIFile sequencer used to play the included demo song.
- Easily extendable menu based panel operation framework.
- Possibility of assigment of the panel knobs to any defined parameter.
- Use of user flash memory to store until 128 sound presets and non volatile configuration data (MIDI device ID, receive MIDI channel, ...)
- Implementation of typical preset dump and request MIDI system exclusive messages.
- Use of object oriented C++ classes in the ColdFire side to make easy the addition of new features, and to promote the reuse of commonly used code.
- Multitasking, object oriented, message-queue and priority based implementation of the different conceptual elements involved.
- Fast access to catalogued parameters by ID using a database-like parameter container (using a fast hash table).
- Easy to change mapping tables to assign MIDI controllers to internal parameters.

It is found in the folder Chameleon.sdk/src/examples/monosynth

5.2 Tutorials

5.2.1 DSP Introductory Tutorials

For those which are not experienced with the Motorola DSPs or with the DSP programming in general, there is an excellent set of tutorials made by Motorola to get hands on on the DSP56300 programming. It comprises 8 exercices which illustrate the DSP addressing modes, implementing FIR filters, advanced use of the Arithmentic Logic Unit and advanced instructions, all with source code included. This tutorial can be found in

the Chameleon.sdk/doc/dsp/other and it is compressed by the files Onyxlabs.pdf and labscode.zip .

5.2.2 An audio level meter for the Chameleon

On this tutorial you will convert the Chameleon into a simple audio level meter. Although this tutorial comes together with the rest of supplied sample code, it also comprises a step by step tutorial, from the algorithm design with the help of Scilab, to the DSP simulation and final implementation and debugging on the harware. It is found on the folder Chameleon.sdk/src/examples/levelmeter

Appendix

A SDK Documentation Index

Following is the complete index for the documentation that can be found in the Chameleon SDK. All the files in the Chameleo.sdk/doc directory are listed in the following table:

File	Contents
doc	
├--chameleon	
│ ├-STD001.pdf	Chameleon Applications Programmer's Guide (This Document)
│ ├--STD002.pdf	Chameleon API Programmer's Reference
│ └-Chameleon Overview.pdf	Brief Introduction to the Chameleon
├--coldfire	Scilab Program Files
│ ├--5206e_um.pdf	ColdFire User's Manual
│ └--cfref_man	ColdFire Programmer's Reference Manual
├--dsp	
│ ├--DSP56300FM.pdf	DSP56300 Family Manual
│ ├--DSP56303UM.pdf	DSP56303 User's Manual
│ ├--**DSP563CCC**.pdf	DSP C Compiler Manual
│ ├--DSPASMRM.pdf	DSP Assembler Reference Manual
│ ├--DSPLINKRM.pdf	DSP Linker Reference Manual
│ ├--DSP56SIMUM.pdf	DSP Simulator User's Manual
│ ├--DSPS56TOOLSUM.pdf	Motorola Suite56™ Tools User Manual
│ ├--appnotes	Several Motorola DSP Application Notes
│ └--other	
│ ├--alpha_inst_ref.pdf	Assembler instruction set for the DSP56300 Family, grouped by name
│ ├--dct.pdf	8x8 DCT Algorithm on Motorola DSP56300
│ ├--DrBob563.zip	Code examples for the DSP5630X
│ ├--Instr-Ref2.pdf	Assembler instruction set for the DSP56300 Family, grouped by function
│ ├--labscode.zip	Code for the Onyxlabs.pdf tutorial
│ └-Onyxlabs.pdf	Several introduction tutorials for the DSP56300
├--gnu	
│ ├-as.pdf	GNU Assembler Manual
│ ├-bdf.pdf	Binary File Descriptor Library

File	Contents
├--binutils.pdf	GNU Binary Utilities
├--cpp.pdf	GNU C Preprocessor Manual
├--gasp.pdf	GNU Assembly Preprocessor Manual
├--gcc.pdf	GNU Compiler Collection Manual
├--ld.pdf	GNU Linker Manual
├--libc.pdf	Cygnus C Support Library Reference
├--libm.pdf	Cygnus Math Library Reference
└--make.pdf	GNU Make Manual
├--midishare	
├--MidiShare.pdf	MidiShare Developer Documentation
└--Player2.0.pdf	MidiShare Player reference Library
├--rtems	
└--c_user.pdf	RTEMS C User's Guide
└--scilab	
├--intro.pdf	Introduction to Scilab
├--manual.pdf	Scilab Reference Manual
└--signal.pdf	Signal Processing with Scilab

Appendix

B MIDI Implementation Chart

Function	Transmitted	Recognized	Remarks
Basic Channel			
Default	**depends**[1]	**depends**[1]	See Notes
Changed	1-16	1-16	
Mode			
Default	**depends**[1]	**depends**[1]	See Notes
Messages	o	o	
Altered	o	o	
Note Number	0-127	0-127	See Notes
Velocity			
Note ON	0-127	0-127	See Notes
Note OFF	0-127	0-127	
After Touch			
Key	o	o	See Notes
Channel	o	o	
Pitch Bender	o	o	See Notes
Control Change	0-127, value 0-127	0-127, value 0-127	See Notes
Program Change	0-127	0-127	See Notes
System Exclusive	o	o	See Notes
System Common			
Song Position	o	o	See Notes
Song Select	o	o	
Tune Request	o	o	
System Real Time			
Clock	o	o	See Notes
Commands	o	o	
Aux. Messages			
Local ON/OFF	o	o	See Notes
All Notes OFF	o	o	
Active Sensing	o	o	
Reset	o	o	

Notes

Chameleon is a fully programmable device and therefore it is capable of receiving and transmiting any type of MIDI data, at the sole discretion of the developer.

depends[1]: The application running in the Chameleon is responsible of setting the default value.

Mode 1: OMNI ON, POLY
Mode 2: OMNI ON, MONO
Mode 3: OMNI OFF, POLY
Mode 4: OMNI OFF, MONO

o : Yes
x : No