soundart

TECHNICAL DOCUMENT
STD002

chameleon api programmer's reference

revision 2 | 07.2002
chameleon S.D.K. v1.2

Table of Contents

0 Introduction 4
0.1 Conventions 4

1 Macros and Specific Functions 6
1.1 Macros for Debugging 6
1.2 Floating and Fixed Point Conversion Utilities 7

2 Panel API 9
2.1 Constants in panel.h 15
2.2 Constants in panel01.h 16

3 FLASH API 17

4 DSP API 21

5 Information API 27

Chapter

0 Introduction

This document is a complete reference for the hardware specific functions found in the libraries of the Chameleon Software Development Kit (SDK), grouped by specific drivers, which compose the Chameleon Application Programming Interface (API). This reference is intended to be used together with the Chameleon Applications Developer's Guide and the rest of the documentation provided with the Chamelon SDK.

The following libraries are explained in detail:

- Chameleon Macros and utility functions.
- Panel Driver API.
- FLASH Driver API.
- DSP Driver API.
- Information Driver API.

The current version of this document covers the Chameleon model #01.

For the latest version of this document, as well as all other SDK components, visit www.soundart-hot.com.

0.1 Conventions

This is a technical document. In order to read it easily, you should be familiar with the typographical conventions employed. Furthermore, most of the material in this reference pertains to the C programming language, and you should ensure that you are familiar with this before attempting to implement the commands explained within.

- The `Courier` font is used for all programming code, except for isolated words.
- **Functions** are usually named using lowercase letters divided by the underscore character, such as: `a_function_name`
- **Constants** are usually named with uppercase letters divided by the underscore character, as in: `PANEL01_SYSTEM_CONSTANT`

- **Numbers** are subject to the following considerations:
- Integer and floating point numbers are written in the normal ways. Examples are 1, 7, 3.142, and so on.
- Hexadecimal numbers are written in the form 0x000000, as in C code. Examples are 0x7F, 0x312A4B, 0xFFFFFF and so on.
- Binary numbers are written in the form %00000000 as in C code. Examples are %01, %01001010 and so on.

Functions are presented in a similar way to the following:

```
some_function_name
```

Declaration

C prototype of the function:

```
int some_function_name(int some_param …);
```

Return value

A description of the value returned by the function.

Parameters

- **some_parameter** : Definition of what the parameter means.

Description

An extended definition of what the function does, how it does it, and any other notes that may be relevant to the programmer.

Chapter

1 Macros and Specific Functions

Following is the listing of the macros and specific functions provided in the header file:

\Chameleon.sdk\include\chameleon\chameleon.h

These utilities are available to the programmer to perform debugging and fixed point conversion tasks. In addition to these utilities, this header file also includes the remaining header files for each Chameleon library, so including chameleon.h is equivalent to include also the rest of the libraries.

1.1 Macros for Debugging

These macros are useful to display messages in the Toolkit terminal window. When an application is compiled in debug mode, the functions behave as described below. When compiling an application for release mode, the preprocessor does not assign any action to these macros and so they will not be executed during program execution.

TRACE ( )

Description

This macro is equivalent to the standard printf function, but it only will print in debug mode. The argument formatted string is sent to the Chameleon serial port and printed in the Toolkit terminal.

ASSERT ( )

Description

This macro evaluates its argument. If the result is 0, the macro prints a diagnostic message and aborts the program. If the condition is nonzero, it does nothing.

1.2 Floating and Fixed Point Conversion Utilities

Since the DSP operates with 24 bit words in fixed point format to represent fractional data[1], and the ColdFire uses 32 bit words floating point data for fractional calculations, some kind of translation is needed for the two processors to communicate. The following conversion routines between both formats are provided to simplify the programmer's implementation of the communication routines between both processors.

fix_to_float

Declaration

```
static __inline float fix_to_float(rtems_unsigned32 n)
```

Description

This function converts the 24 bit fixed point parameter n (sign extended to 32 bits) into a 32 bit IEEE format floating point value.

float_to_fix

Declaration

```
static __inline rtems_unsigned32 float_to_fix(float f)
```

Description

This function converts the 32 bit IEEE format floating point parameter f into a 24 bit fixed point value (sign extended to 32 bits). The result is truncated (no rounding is performed).

[1] For a detailed reference on the fixed point fractional data used by the DSP, please refer to the Motorola Application Report APR/3, Fractional And Integer Arithmetic using the DSP56000 family.

float_to_fix_round

Declaration

```
static __inline rtems_unsigned32 float_to_fix_round(float f)
```

Description

This function converts the 32 bit IEEE format floating point parameter f into a 24 bit fixed point value (sign extended to 32 bits). The result is rounded.

Chapter

2 Panel API

The front panel controls of the Chameleon are accessed uing the Panel library. All interactive controls (except the power switch) can be managed from the application employing the function calls below.

The header file for these functions is stored in:

\Chameleon.sdk\include\chameleon\panel.h

panel_init

Declaration

```
int panel_init(void);
```

Return value

Returns a non zero value with the opened panel handle. If the function fails, it returns zero.

Parameters

None.

Description

This function initializes the panel driver and returns a handle to it, which is used in all the subsecuent Panel functions calls. Before to open the Panel by calling panel_init, no other panel function may be called.

panel_exit

Declaration

```
rtems_boolean panel_exit(int ref);
```

Return value

Returns TRUE if the Panel driver was closed succesfully, and FALSE if the function fails or the driver was already closed.

Parameters

- **ref** : Handle of the panel to close.

Description

This function releases the handle to the panel obtained with panel_init. Once this handle is released, any attempt to use any panel function again without re-initialising it will cause that function to return an error.

panel_out_lcd_clear

Declaration

```
rtems_boolean panel_out_lcd_clear (int ref);
```

Return value

Returns TRUE if the function suceeds, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel to close.

Description

It clears the LCD screen of the Chameleon. Any displayed text is erased. This function can be used any time to make sure that there are no random characters left over from whatever was being displayed previously.

panel_out_lcd_print

Declaration

```
rtems_boolean panel_out_lcd_print(int ref, rtems_unsigned8 row,
                                  rtems_unsigned8 col, char *text);
```

Return value

Returns TRUE if the function suceeds, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **row**: LCD row to start printing on. Possible values are 0 (Upper row) or 1 (Lower row). Greater values are ignored.
- **col**: LCD column to start printing on. Values greater or equal than PANEL_LCD_MAX_LINE_LEN are ignored.
- **text**: pointer to the string to be displayed.

Description

This function displays the string pointed to by text on the LCD, starting at the (row, col) character. If the stringt has a length greater than PANEL_LCD_MAX_LINE_LEN, extra characters are ignored.

A special case is when the string contains the characters 0x01 to 0x08. These characters may be redefined by the programmer, using the function panel_out_redefine(). In this case the user defined characters are displayed.

panel_out_lcd_redefine

Declaration

```
rtems_boolean panel_out_lcd_redefine(int ref, rtems_unsigned8 code,
                                     const rtems_unsigned8 *data);
```

Return value

Returns TRUE if the function suceeds, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **code**: Specifies the user character to redefine. Possible values are 0 to 7.
- **data**: Points to the character pixel map, which is an array of 8 rtems_unsigned8 type elements, each containing the character's 5 row dots in his LSB bits. The LCD characters are 5 pixels wide by 8 pixels height, so only the 5 LSB of each array element and 8 elements are used. The data can be declared as follows:

```
static rtems_unsigned8 lcd_symbol [8] =
{
   0x01,   /* 00000001 */
   0x03,   /* 00000011 */
   0x07,   /* 00000111 */
   0x0f,   /* 00001111 */
   0x07,   /* 00000111 */
   0x03,   /* 00000011 */
   0x01,   /* 00000001 */
   0x00    /* 00000000 */
};
```

Description

Redefines the specified LCD user character. If this character is already being displayed on the LCD when calling this function, it will change according to the new definition.

When defining a new character, the hexadecimal values in the data array are converted to binary form and treated as on/off values for the actual pixels in the LCD. Because no character in Model #01 of the Chameleon may be more than 5 pixels wide, this means that only values between 0x00 (%00000000) and 0x1F (%00011111) are useful. Higher bits are ignored.

panel_out_led

Declaration

```
rtems_boolean panel_out_led(int ref, rtems_unsigned32 led_bits);
```

Return value

Returns TRUE if the function suceeds, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **led_bits**: bit wise led values (on/off). See the constants in panel01.h for the currently meaningful values. Other bits apart from these are ignored.

Description

Turns on/off the specified panel LED. All the LEDs can be operated at once.

panel_in_new_event

Declaration

```
rtems_boolean panel_in_new_event(int ref, rtems_boolean wait);
```

Return value

Returns TRUE if the function executes successfully, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **wait**: specifies if the function has to wait until a new panel event is received or exit immediately.

Description

This function asks the panel driver if a new event has been received from the Panel. It is possible to block the calling task until a new event is received by setting the wait parameter to TRUE. In this case, the calling task will be blocked in a cooperative way with the rest of application's tasks, wich will continue their normal execution.

Once a new event is received, the calling task will be unblocked and will continue executing.

If this function is called with the wait parameter set to FALSE, it will check for a new incoming panel event and will return immediately. If it executes succesfully, it will return TRUE even if there's not a new event received. The program should check then if there is actually a new event by calling panel_in_potentiometer(), panel_in_encoder(), and panel_in_key().

Possible events from the Front Panel are keys pressed and encoder and potentiometer movements.

panel_in_potentiometer

Declaration

```
rtems_boolean panel_in_potentiometer(int ref,
                                     rtems_unsigned8 *potentiometer,
                                     rtems_unsigned8 *value);
```

Return value

Returns TRUE if the new received panel event correspods to a potentiometer movement, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **potentiometer**: specifies the number of the potentiometer that has been moved. For possible values, see the panel01.h constants definition.
- **value**: specifies the new value of the moved potentiometer in a range from 0 to 127.

Description

Once a new incoming event from the Panel has been received via a panel_in_new_event , this function is used to verify if such event corresponds to a potentiometer movement on the front panel.

panel_in_keypad

Declaration

```
rtems_boolean panel_in_keypad (int ref, rtems_unsigned32 *key_bits);
```

Return value

Returns TRUE if the new received panel event correspods to a key pressed, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **key_bits**: bit wise key values (on/off). See the constants in panel01.h for the currently meaningful values. Other key bits apart from these are ignored.

Description

Once a new incoming event from the Panel has been received via a panel_in_new_event , this function is used to verify if such event corresponds to a key pressed on the front panel.

panel_in_encoder

Declaration

```
rtems_boolean panel_in_encoder(int ref, rtems_unsigned8 *encoder,
                               rtems_signed8 *increment);
```

Return value

Returns TRUE if the new received panel event correspods to a encoder moved, otherwise returns FALSE.

Parameters

- **ref** : Handle of the panel.
- **encoder**: specifies the encoder number that has been moved. In the Chameleon model #01, which has only one encoder, this parameter is allways 1.
- **increment**: specifies the amount of encoder steps actually moved.

Description

Once a new incoming event from the Panel has been received via a panel_in_new_event , this function is used to verify if such event corresponds to a encoder moved on the front panel.

2.1 Constants in panel.h

The following constants to be used by the programmer are defined in the file

\Chameleon.sdk\include\chameleon\panel.h

```
#define  PANEL_LCD_MAX_LINE_LEN 16 // Maximum number of characters
                                   // in a LCD text line
```

2.2 Constants in panel01.h

The header file

\Chameleon.sdk\include\chameleon\panel01.h

contains definitions for the Chamelon model #01 specific panel characteristics. Following is the listing of definitions to keep in mind when programming the Chameleon front panel.

```
/* LED BITS DEFINES: */

#define PANEL01_LED_CTRL3       0x01000000
#define PANEL01_LED_CTRL2       0x02000000
#define PANEL01_LED_CTRL1       0x04000000
#define PANEL01_LED_SHIFT       0x08000000
#define PANEL01_LED_EDIT        0x10000000

/* POTENTIOMETER NUMBER DEFINES: */

#define PANEL01_POT_VOLUME      0x00
#define PANEL01_POT_CTRL1       0x01
#define PANEL01_POT_CTRL2       0x02
#define PANEL01_POT_CTRL3       0x03

/* KEY BITS DEFINES: */

#define PANEL01_KEY_GROUP_UP    0x01000000
#define PANEL01_KEY_PAGE_UP     0x02000000
#define PANEL01_KEY_GROUP_DOWN  0x04000000
#define PANEL01_KEY_PAGE_DOWN   0x08000000
#define PANEL01_KEY_PARAM_UP    0x10000000
#define PANEL01_KEY_VALUE_UP    0x20000000
#define PANEL01_KEY_PARAM_DOWN  0x40000000
#define PANEL01_KEY_VALUE_DOWN  0x80000000
#define PANEL01_KEY_EDIT        0x00010000
#define PANEL01_KEY_PART_UP     0x00020000
#define PANEL01_KEY_SHIFT       0x00040000
#define PANEL01_KEY_PART_DOWN   0x00080000
```

Chapter

3 FLASH API

The FLASH memory of the Chameleon is used to store applications and the permanent data used by them. The FLASH is managed through the FLASH memory library, wich provides with the appropiate functions to perform read and write operations from the application's code.

The header file for these functions is stored in:

\Chameleon.sdk\include\chameleon\flash.h

flash_init

Declaration

```
int flash_init(void);
```

Return value

Returns a non zero value with FLASH memory handle. If the function fails, it returns zero.

Parameters

None.

Description

This function initializes the flash driver and returns its handle, which is used in the all the subsecuent FLASH functions calls. No other FLASH functions may be called before opening the FLASH.

flash_exit

Declaration

```
rtems_boolean flash_exit(int ref);
```

Return value

Returns TRUE if the FLASH driver was closed succesfully, and FALSE if the function fails or the driver was already closed.

Parameters

- **ref** : Handle of the FLASH.

Description

This function releases the FLASH handle obtained with flash_init. Once this handle is released, any attempt to use any FLASH function again without re-initialising it will cause that function to return an error.

flash_read_data

Declaration

```
rtems_boolean flash_read_data(int ref, rtems_unsigned32 offset,
                              rtems_unsigned8 *data,
                              rtems_unsigned32 count);
```

Return value

Returns TRUE if the FLASH driver readed succesfully the data, and FALSE if the function failed.

Parameters

- **ref** : Handle of the FLASH.
- **offset**: memory address offset in bytes from where the read is going to be performed. The valid range of values starts from 0x00000000 up to the value obtained by the flash_get_size() function.
- **data**: Pointer to a data storage DRAM memory block or variable.
- **count**: Number of bytes actually read.

Description

This function attempts to read a block of count bytes from the FLASH memory starting at the address specified by offset and stores it in the DRAM memory block pointed by data.

It is not possible to read the FLASH memory space where the application code itself is stored. Only the memory area which is unused by the application can be readed. To obtain the amount of memory available to read, use flash_get_size().

flash_write_data

Declaration

```
rtems_boolean flash_write_data(int ref, rtems_unsigned32 offset,
                               const rtems_unsigned8 *data,
                               rtems_unsigned32 count);
```

Return value

Returns TRUE if the FLASH driver has written succesfully the data, and FALSE if the function fails.

Parameters

- **ref** : Handle of the FLASH.
- **offset**: memory address offset in bytes where the data is going to be stored. The valid range of values starts from 0x00000000 up to the value obtained by the flash_get_size() function.
- **data**: Pointer to the DRAM memory block or variable which contains the data to be stored.
- **count**: Number of bytes to write.

Description

This function attempts to write a block of count bytes in the FLASH memory starting at the address specified by offset and readed starting at the DRAM memory block pointed by data.

It is not possible to write the memory area where the application code itself is stored. Only the memory area which is unused by the application can be written. To obtain the amount of memory available to write, use flash_get_size().

flash_get_size

Declaration

```
rtems_boolean flash_get_size(int ref, rtems_unsigned32 *size);
```

Return value

Returns TRUE if the FLASH driver obtained succesfully the available FLASH memory size, and FALSE if the function fails.

Parameters

- **ref** : Handle of the FLASH.
- **size**: amount of free FLASH memory available.

Description

This function returns the amount of FLASH memory available to read and write operations, apart from the memory space occupied by the stored application's code.

Chapter

4 DSP API

The Chameleon DSP is managed on the ColdFire side by using the DSP library, which contains the necessary functions to exchange data between them.

The header file for these functions is stored in:

\Chameleon.sdk\include\chameleon\dsp.h

dsp_init

Declaration

```
int dsp_init(int dsp_index, const rtems_unsigned8 *code);
```

Return value

Returns a non zero value with DSP handle. If the function fails, it returns zero.

Parameters

- **dsp_index**: identifier of DSP to be initialized. In the Chameleon model #01, the value of this parameter is allways 1.
- **code**: pointer to the code to be downloaded into the DSP.

Description

This function initializes the DSP driver and downloads the processing code to run on it. DSP source code (if any) is converted into machine code at compile time, and the machine code is then stored as an array of bytes in a C header file. This header file has to be included by the code calling the function to get access to the parameter code.

When this function is called, the DSP code is trasmitted from the Coldfire to the DSP56303 via the HI08 port automatically. Once the function has successfully completed, DSP code execution begins immediately. Before opening the DSP driver by calling dsp_init, no other DSP function may be called.

dsp_exit

Declaration

```
rtems_boolean dsp_exit(int ref);
```

Return value

Returns TRUE if the DSP driver was successfully closed, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.

Description

This function releases the DSP handle obtained with dsp_init. Once this handle is released, any attempt to use any DSP function again without re-initialising it will cause that function to return an error.

dsp_read_data

Declaration

```
rtems_boolean dsp_read_data(int ref, rtems_signed32 *data,
                            rtems_unsigned32 count);
```

Return value

Returns TRUE if the DSP data were readed successfully, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **data**: Pointer to the DRAM memory block or variable to store the readed data.
- **count**: Number of words actually read.

Description

This function reads count 32 bit words from the DSP's HI08 Host port and stores it in the variable or memory block pointed by data. Values readed are 24 bit DSP words sign extended to 32 bit. The calling task is blocked in a cooperative way (other tasks continue its normal execution) until all

the words are read or after 3 seconds after the calling if all the words were not read.

dsp_write_data

Declaration

```
rtems_boolean dsp_write_data(int ref, const rtems_signed32 *data,
                             rtems_unsigned32 count);
```

Return value

Returns TRUE if the data were written successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **data**: Pointer to the DRAM memory block or variable from which to get the data to write to the DSP.
- **count**: Number of words to write.

Description

This function writes the count 32 bit words stored in the variable or memory block pointed to by data to the DSP's HI08 Host port. The DSP will only read the 24 LSB bits, so the 8 MSB are effectively meaningless. The calling task is blocked in a cooperative way (other tasks continue its normal execution) until all the words are written.

dsp_write_command

Definition

```
rtems_boolean dsp_write_command(int ref, rtems_unsigned8 command,
                                rtems_boolean wait);
```

Return value

Returns TRUE if the command is written successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **command**: Number of DSP command (interrupt vector address) to write divided by two. Possible values are from 0x00 to 0x7F.
- **wait**: If TRUE, it instructs the function to wait until the command is actually written, otherwise the functions returns immediately.

Description

This function is used to write Host Commands to the DSP's HI08 Host port. By writing a command on the DSP, the calling routine causes the DSP to execute an interrupt handling routine whose vector corresponds to the value specified in the command parameter multiplied by two. If the parameter wait is set to TRUE, the calling task will be blocked in a cooperative way (other tasks continue its normal execution) until the command is actually written to the DSP.

dsp_write_flag0

Declaration

```
rtems_boolean dsp_write_flag0(int ref, rtems_boolean flag);
```

Return value

Returns TRUE if the flag is written successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **flag**: Boolean value of the Flag to be written.

Description

This function writes the general purpose flag HSR_HF0 on the DSP's Host Port HSR register. Its value is set if flag is TRUE, or zero otherwise.

DSP flags 0 and 1 have no intrinsic meaning to the DSP. They are a convenience, allowing the programmer to pass information from the Coldfire to the DSP. It is the programmer's responsibility to write appropriate DSP code to monitor and respond to the status of these two flags.

dsp_write_flag1

Declaration

```
rtems_boolean dsp_write_flag1(int ref, rtems_boolean flag);
```

Return value

Returns TRUE if the flag is written successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **flag**: Boolean value of the Flag to be written.

Description

This function writes the general purpose flag HSR_HF1 on the DSP's Host Port HSR register. Its value is set if flag is TRUE, or zero otherwise.

Neither of flags 2 or 3 has any special meaning to the Coldfire or RTEMS. Similar to the HSR_HF0 and HSR_HF1 flags, they can be used to pass status information from the DSP back to the Coldfire.

dsp_read_flag2

Declaration

```
rtems_boolean dsp_read_flag2(int ref, rtems_boolean *flag);
```

Return value

Returns TRUE if the flag is read successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **flag**: Pointer the Boolean value to read the flag.

Description

This function reads the general purpose flag HCR_HF2 on the DSP's Host Port HSR register. The variable pointed by flag will be TRUE if this flag is set, or FALSE otherwise.

dsp_read_flag3

Declaration

```
rtems_boolean dsp_read_flag3(int ref, rtems_boolean *flag);
```

Return value

Returns TRUE if the flag is read successfully to the DSP, otherwise it returns FALSE.

Parameters

- **ref** : Handle of the DSP.
- **flag**: Pointer the Boolean value to read the flag.

Description

This function reads the general purpose flag HCR_HF3 on the DSP's Host Port HSR register. The variable pointed by flag will be TRUE if this flag is set, or FALSE otherwise.

Chapter

5 Information API

The Information driver allows the programmer to get specific information related to the Chameleon hardware, such as the device's serial number, the firmware version and the model number. This is the only driver that needs not to be previously initialized before using its functions.

These functions are defined in the file:

\Chameleon.sdk\include\chameleon\info.h

info_get_serial_number

Declaration

```
rtems_boolean info_get_serial_number(char serial_number[11]);
```

Return value

Returns TRUE if the device serial number is read successfully, otherwise it returns FALSE.

Parameters

- **serial_number**: String to store the device serial number.

Description

This function returns the device serial number. This number is the same that is printed on the device's case and that is displayed on the LCD when booting in wait mode and pressing the "Shift" key.

info_get_model

Declaration

```
rtems_boolean info_get_model(rtems_unsigned8 *model);
```

Return value

Returns TRUE if the device model's identifier is read successfully, otherwise it returns FALSE.

Parameters

- **model**: Variable store the device's model identifier.

Description

This function returns the device model identifier. The purpose of this function is to allow compatibily with future models, by letting the application to know which header files has to include to work properly.

At the time of writing, only one model of the Chameleon is available and thus there is only one set of include files for the hardware configuration.