SH4 Debug Interface Guide

Cross Products Limited

Legal Notice

IMPORTANT

This equipment is Class A Information Technology Equipment (ITE) as defined in EN55022:1994.

Warning

This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

NOTE: THIS EQUIPMENT HAS BEEN TESTED AND FOUND TO COMPLY WITH THE LIMITS FOR "CLASS A" DIGITAL DEVICES, PURSUANT TO PART 15 OF THE FCC RULES. THESE LIMITS ARE DESIGNED TO PROVIDE REASONABLE PROTECTION AGAINST HARMFUL INTERFERENCE WHEN THE EQUIPMENT IS OPERATED IN A COMMERCIAL ENVIRONMENT. THIS EQUIPMENT GENERATES, USES, AND CAN RADIATE RADIO FREQUENCY ENERGY AND, IF NOT INSTALLED AND USED IN ACCORDANCE WITH THE INSTRUCTION MANUAL, MAY CAUSE HARMFUL INTERFERENCE TO RADIO COMMUNICATIONS. OPERATION OF THIS EQUIPMENT IN A RESIDENTIAL AREA IS LIKELY TO CAUSE HARMFUL INTERFERENCE IN WHICH CASE THE USER WILL BE REQUIRED TO CORRECT THE INTERFERENCE AT HIS OWN EXPENSE.

CHANGES OR MODIFICATIONS NOT EXPRESSLY APPROVED BY CROSS PRODUCTS LTD. COULD VOID THE USER'S AUTHORITY TO OPERATE THE EQUIPMENT.

SH4 Debug Interface Guide

Revision History

First release April 2000. CodeScape version 3.0.0

Cross Products Ltd
23 The Calls, Leeds, West Yorkshire, LS2 7EH
telephone: +44 113 242 9814
facsimile: +44 113 242 6163
www.crossprod.co.uk
email sales: enquiry@crossprod.co.uk
email support: support@crossprod.co.uk

Contents

About this guide

“ASE and Extended debug stubs” provides detailed background information about the two debug stubs and their differences.

“*Interrupts and Exceptions*” provides details of how interrupts and exceptions are implemented under different debugging environments.

“Communications Channels” provides an overview of channels and how they are accessed from the target and host using ASE BIOS services and the CPDIAL library respectively.

“ASE BIOS services” describes ASE BIOS calls and channel interrupts used to access channels from the target and gives code examples.

“CPDIAL Library Reference” describes the Debug Interface Adapter Library, including the CDIAL C++ class definitions, and how to use its functions to access channels from the host.

ASE and Extended debug stubs

The debug stub has two parts, the ASE debug stub and the Extended debug stub. Two stubs are required because ASE memory is limited to 1kbyte and is insufficient to provide all the required functionality. The Extended debug stub, due to its size, must be resident in external RAM.

The two stubs and their environment requirements are discussed in this chapter.

The ASE debug stub

The ASE debug stub resides in the SH4’s 1kbyte of ASE memory. This memory cannot be physically read or written to by the application code, it can only be accessed in ASE mode by the debug tools. The ASE debug stub is a minimal debug stub loaded into ASE RAM after a target system reset. The purpose of the ASE debug stub is to facilitate basic debugging, system initialisation and the loading of the larger and functionally complete Extended debug stub. Once the Extended debug stub is loaded the ASE debug stub is used simply to chain ASE exceptions to handlers within the Extended debug stub. The ASE debug stub and ASE RAM are transparent to the application.

The Extended debug stub

The Extended debug stub resides in RAM and requires 16kbytes of memory. This debug stub contains all the necessary functionality to provide comprehensive debugging of the target. The debug stub memory is not protected from corruption by the application code, thus the application programmer must not under any circumstance attempt to utilise the memory space allotted to the Extended debug stub. An errant piece of application code can write over the Extended debug stub causing it to fail. Under these circumstances the debug stub (and debug session in progress) must be reloaded and restarted. The Extended debug stub has its own private stack within the boundaries of the allotted 16kbytes of RAM and thus does not require use of the application program’s stack.

The location of the Extended debug stub is defined either by the Boot ROM code, or by the settings in the Start-up configuration dialog of CodeScape.

The Extended debug stub can also be identified by the three instructions at the start of the 16kbyte section, which are always BRK, RTE, NOP; instruction codes 0x003B, 0x000B, and 0x0009.

Byte sequence:

- 0x3B, 0x00, 0x0B, 0x00, 0x09, 0x00 - little endian Extended debug stub
- 0x00, 0x3B, 0x00, 0x0B, 0x00, 0x09 - big endian Extended debug stub

Functional differences of the ASE and Extended debug stubs

The table below lists the differences in functionality of the two stubs.

Function	Extended debug stub	ASE debug stub	Comments
Read/Write memory	yes	yes	Extended: Optimized for speed. ASE: Optimized for size.
Write memory cache coherency maintained.	yes	yes	Extended: Handled by the Extended debug stub. ASE: Handled by the DA (much slower).
Read/Write General purpose registers.	yes	yes	
Read/Write Floating point registers	yes	yes	
Read/Write UBC registers	yes	yes	Extended: Optimized for speed. ASE: Achieved by multiple memory R/W accesses (much slower).
Read/Write HBC registers	yes	yes	Extended: Optimized for speed. ASE: Achieved by multiple memory R/W accesses (much slower).

Function	Extended debug stub	ASE debug stub	Comments
Read/Write MMU registers.	yes	yes	Extended: Optimized for speed. ASE: Achieved by multiple memory R/W accesses (much slower). ASE: Store queue info not available.
Default exception handler at VBR.	yes	no	Extended: Located wholly within the boundaries of the 16kbytes allotted.
Catch exceptions at VBR + 0x100.	yes	yes	Extended: Uses default exception handler. ASE: Uses SH4's trace mechanism.
Catch exceptions at VBR + 0x400.	yes	yes	Extended: Uses default exception handler. ASE: Uses SH4's trace mechanism.
Catch interrupts at VBR + 0x600.	yes	yes	Extended: Uses default exception handler. ASE: Uses SH4's trace mechanism.
Exception passback to stub.	yes	no	
Profiling, statistical sampling.	yes	no	
Profiling, tracing	yes	no	
BIOS calls for virtual channels.	yes	no	
BIOS calls for utility functions.	yes	no	
Supports little endian systems.	yes	yes	Extended: DA detects the endianness of the target and loads a little endian Extended debug stub when required. ASE: ASE RAM is always big endian.
Supports big endian systems	yes	yes	Extended: DA detects the endianness of the target and loads a big endian Extended debug stub when required. ASE: ASE RAM is always big endian.
Require system memory to be present and configured.	yes	no	Extended: Requires 16kbytes (anywhere in RAM). ASE: Required no RAM at all.
Facilitates debugging of code in ROM.	yes	yes	

Function	Extended debug stub	ASE debug stub	Comments
Facilitates debugging of code in RAM.	yes	yes	
Facilitates the use of software breakpoints	yes	yes	
Facilitates the use of hardware breakpoints	yes	yes	
ROMless system 'bring up'.	yes	yes	
RAMless system 'investigation'.	no	yes	
ASE mode used for debugging.	yes	yes	Extended: Vectored into from ASE RAM. ASE: Used inherently.
Hitachi SH4 FPU exception handled.	yes	no	Extended: Transparent to user (code execution slowed down). ASE: Exception reported (handled by CodeScape).
Stub memory protected for application code.	no	yes	

Optimization of the stubs

The ASE debug stub must fit entirely into 1kbyte (512 instructions), for this reason all of the ASE debug stub's functions are optimized for size. In some cases functionality is delegated from the ASE debug stub to the Debug Adapter (DA).

An example of this is the 'Read Context HBC'. This is a command issued by CodeScape to request that the values of all of the HBC's (hardware break controller) memory mapped registers are returned.

In the Extended debug stub, a protocol exists for this purpose. The DA requests the HBC's context and the Extended debug stub replies by reading the required registers and returning them in a single transaction.

In the ASE debug stub no such protocol exists between the DA and the ASE debug stub, however the same result is achieved by the DA issuing a series of 'Read Memory' commands to read each of the individual registers one by one.

The DA and Extended debug stub performs the HBC context read in the most efficient (fastest) method possible whereas the DA and the ASE debug stub performs the HBC context read using the minimum amount of code in the ASE debug stub. The net result is that the Extended debug stub runs much faster than the ASE debug stub.

Exception handling in the ASE debug stub

The ASE debug stub uses the SH4's Branch Trace mechanism to allow exceptions and interrupts to be caught. The Branch Trace mechanism is part of the SH4's on-chip hardware debugging tools. This allows the ASE debug stub to be entered, via an ASE Trace exception, when a particular condition is met.

The Trace Branch mechanism is set up to cause the stub to be entered when either; an exception to VBR + 0x100, an exception to VBR + 0x400, an interrupt to VBR + 0x600 or an RTE instruction is executed. When the stub is entered due to the one the above Branch Trace events occurring the following action is taken:

VBR + 0x100	Code is halted and the event is reported to the DA.
VBR + 0x400	Code is halted and the event is reported to the DA.
VBR + 0x600	Program execution is resumed (under control of the DA).
RTE	Program execution is resumed (under control of the DA).

The Branch Trace events VBR + 0x600 and RTE are usually not required as far as the user is concerned, but they must be handled because the SH4's Branch Trace mechanism does allow individual branch types to be selected. Thus for VBR + 0x600 and RTE, the DA instructs the ASE debug stub to simply return program control back to the application program.

If the ASE debug stub is being used to debug an application that uses interrupt VBR + 0x600 and its associated RTE, there will be a time penalty due to the 'filtering' effect of processing this unwanted exception.

When using the ASE debug stub for debugging, if you do not want the ASE debug stub to catch any exceptions, then it is possible to turn off the Branch Trace mechanism entirely. This is achieved using CodeScape (from a memory window) to write a single byte 0x00 to the 'Trace Memory Control Register' at address 0xFF2000BC. It should be noted that without the Branch Trace mechanism the ASE debug stub will no longer be able to catch any exceptions (including ones such as Address Error etc.) and in CodeScape you will be limited to using hardware and software breakpoints and basic trace functions such as single stepping.

Exception handling in the Extended debug stub

The Extended debug stub handles exceptions and interrupts by installing a default exception handler which sets up the VBR register accordingly. Alternatively, you can use your own application exception handler, *see “Interrupts and Exceptions” on page 11*.

Debug stub cache usage

Caching during BIOS calls

When a BIOS call is made the debug stub runs from the SH4’s P1 area (cacheable). If the SH4’s cache is enabled then the speed of the BIOS call is improved due to the debug stub being in a cacheable area. The debug stub does not implicitly enable the cache during a BIOS call.

Cache coherency for the 'Write Memory' command

Cache coherency for writing to the SH4's memory is maintained for both the ASE debug stub and the Extended debug stub. The basic method used to achieve this is:

- The CCR (cache control register) is put into 'write through' mode before actually writing to the SH4's memory, write to the memory, then return the CCR to its original state.
- Before 'resuming' running the application code (after a memory write), the instruction cache is flushed.

For the Extended debug stub

The manipulation of the CCR for the Extended debug stub is done entirely in the Extended debug stub's 'Write Memory' command. If a memory write to the SH4 has been issued by the DA then the following sequence of events is executed:

- Prior to the memory write the current CCR (cache control register) value is saved.
- The cache is put into write through mode.
- The memory write is carried out.
- The CCR is returned to its original value.

At the end of the debug session (the series of commands issued when in the debug stub prior to resuming the application code) the instruction cache is invalidated.

For the ASE debug stub

For the ASE debug stub, the CCR is manipulated by the DA issuing 'Read Memory' and 'Write Memory' commands to read and write the CCR register before and after issuing the actual 'Write Memory' command itself. Using this method the ASE debug stub's 'Write Memory' command does not need any code to handle cache coherency, however, the performance is degraded due to the overhead of the DA issuing the additional commands.

Interrupts and Exceptions

This section deals with issues concerning exception and interrupt handling on the target and the interaction and interrupt requirements of the debug stub.

Exception handling with or without a Boot ROM

Halt or Resume after stub load

Once the Extended debug stub has loaded, the Debug Adapter (DA) instructs the debug stub to either resume execution of the Boot ROM (if one is installed), or set up a default environment and wait for the user to perform some action such as downloading a program. This is controlled by the option: *Halt after stub load* on the *Start-up options* dialog box in CodeScape.

The debug stub implements exceptions using different methods depending on this condition.

Resume (debugging with a Boot ROM)

This mode exists to allow program execution to resume back to the Boot ROM after the Extended debug stub is loaded. This is achieved with the minimum of disruption to the state of the target microprocessor context.

However the following changes will have been made to the context after resuming:

- The DBR register is loaded with the default debug stub exception handler.

Halt (debugging without a Boot ROM)

This mode exists to allow the Extended debug stub to be loaded and then the context to be set up with default settings to allow a debugging session to commence. In this mode CodeScape does not pass program control back to the Boot ROM, instead, control remains within the debug stub monitor.

- The DBR register is loaded with the default debug stub exception handler.
- The VBR register is loaded with the default debug stub exception handler.
- The stack pointer is loaded at 0x0d000000.
- The status register block bit is cleared to 0.

Exceptions during debugging with a Boot ROM

When *'resume after stub load'* mode has been selected, the debug stub does not implement a exception or interrupt handler of its own and it does not alter or set up the VBR register. Thus the debug stub makes no demands of the application code and does not introduce any time penalties by running debug stub exception handler code. However, this does mean that the application code must implement its own handler if exceptions or interrupts are to be used by the application code itself.

To allow the debug stub the ability to process exceptions (not interrupts, *see "Interrupts during debugging" on page 14.*) in this mode, the debug stub incorporates a passback facility to allow unhandled exceptions to be caught and processed by the debug stub. To implement the passback facility the application code exception handler must call the debug stub using the following sequence of instructions.

For exceptions (VBR + 0x100 and VBR + 400) the passback call takes the form:

```
BRK
RTE
RTE
NOP
```

This special sequence causes the debug stub to be entered via the 'BRK' instruction and the double 'RTE' immediately after the break indicates to the debug stub that this is an exception passback call.

This sequence should be added to the end of the application code exception handler to allow the debug stub to process any unhandled exceptions. The 'NOP' at the end is not strictly required but can be added to inhibit compiler or assembler errors. You should use the passback sequence where you would normally put code to deal with unhandled exceptions. Where an exception has been handled correctly by the application code, then an 'RTE' must be executed rather than the passback sequence.

Notes:

1. *After the debug stub processes an unhandled exception, the program execution is returned to the interrupted code, control is not returned to the application exception handler. Thus no attempt is made to try and execute the 'RTE RTE' element of the passback sequence.*
2. *Some assemblers and compilers may not allow an RTE opcode following an RTE opcode. In this event the following sequence could be used:*

```
BRK
RTE
DC.W       0x002b
NOP
```

3. *Unhandled exceptions passed back to the debug stub are reported to CodeScape and this in turn is interpreted and reported as specified by CodeScape.*
4. *The application code must take care to ensure the SR.BL block bit is handled correctly to ensure exceptions are accepted (see the Hitachi SH4 hardware manual for further details).*
5. *For the passback facility to function correctly the application code must not change the contents of the EXPEVT register prior to issuing the passback call.*

Exceptions during debugging without a Boot ROM

When *'halt after stub load'* mode has been selected, the debug stub implements a default exception and interrupt handler by initialising the VBR register to point at the debug stub default handler.

The default handler allows all exceptions and interrupts to be caught by the debug stub and in turn reported by CodeScape. This allows application code to be debugged without the need for an application code handler to catch exceptions such as 'address error' etc.

The application code can at any time install its own handler by changing the contents of the VBR. Once this is done the debug stub default handler will no longer function but application code can utilise the passback method as described in the previous section to allow unhandled exceptions to be handled by the debug stub.

If the VBR is changed by the application code, the debug stub will make no attempt to restore the VBR to point at the debug stub default handler. However the application code can restore the VBR to the default value to make the default handler functional again.

Notes:

1. *If 'halt after stub load' debugging mode is selected the SR.BL block bit defaults to 0.*

Interrupts during debugging

An additional specific passback sequence is available to allow unhandled interrupts (not exceptions) to be captured and reported to CodeScape. To implement this passback facility the application code interrupt handler must call the debug stub using the following sequence of instructions.

```
BRK
RTE
RTS
NOP
```

The above sequence should be implemented in a similar manner to the exception passback sequence described above. This sequence must only be used to passback unhandled interrupts for the interrupt handler at the address VBR + 0x600.

Caveats and limitations

Use of the HBC registers

CodeScape makes use of the HBC unit to implement hardware breakpoints, for this reason the application code must not attempt to access the HBC's registers.

Debugging exception handlers using CodeScape

It is possible to 'lose' hardware breakpoints when debugging application exception handlers. This occurs if the handler routine being debugged does not allow for nested exceptions to occur. A nested exception is required in this instance to allow the HBC exception to be accepted and the debug stub default handler to be called. It is recommended that software breakpoints are used where possible when debugging exception handlers.

Consult the Hitachi SH4 hardware manual for information on allowing nested interrupts to occur, however the following notes outline the basic methodology. On entering the application exception handler:

1. Save the SPC to a temporary location.
2. Save the SSR to a temporary location.
3. Clear the SR.BL bit to 0 (this now allows exceptions or interrupts to occur).
4. Execute the application exception handler code.
5. Set the SR.BL bit to 1. (this inhibits exceptions or interrupts from occurring when restoring the SPC and SSR).
6. Restore the SSR to the original value from the temporary location.
7. Restore the SPC to the original value from the temporary location.
8. RTE

Notes:

1. *If an interrupt occurs within the exception handler prior to the SR.BL bit being cleared, then it will be held until the SR.BL bit is cleared.*
2. *If an exception occurs within the exception handler prior to the SR.BL bit being cleared or after it has been set, then it will cause a manual reset to occur causing program execution to transfer to the reset vector (highly undesirable). This situation will occur if the exception handler contains a programming error in the code such as an instruction address error. Care must be taken with writing application exception handlers to ensure no such errors exist, specifically when the SR.BL bit is set.*

Performance loss when handling SH4 FPU exceptions

If the enable bit for underflow, overflow and inexact in the FPSCR registers is set, the processor reports an exception for all operations that can possibly generate the desired exception, regardless of whether an error is detected. It is the exception handler's responsibility to differentiate between real and false exceptions by masking together the cause and enable bits of the FPSCR register.

To alleviate this problem the Extended debug stub checks these bits and ignores false exceptions. Processing exceptions for every floating point operation can seriously impact on the performance of the processor.

The ASE debug stub cannot process the exceptions and relies on CodeScape to handle the problem. The process of stopping the processor and waiting for the debugger to service the exception produces an even larger degradation in performance.

Communications Channels

About channels

Channels provide a means of fast communication between the host and the target. There are four high-speed bi-directional channels allowing communication between host and target applications at speeds in excess of 500kbytes/s. Each channel may be read from and written to independently either a byte at a time or in blocks.

Channels are implemented via the JTAG interface from the target to the Debug Adapter (DA) and via Ethernet from the DA to your computer. The DA has eight 128kbyte buffers to provide an input and output buffer for each channel to ensure uninterrupted communication.

The following chapters describe in detail the ASE BIOS services and DIAL library functions and give code examples where appropriate.

NOTE: *At the time of writing, you cannot use channels with Microsoft® Windows® CE.*

Channel access

Target access

Channels are accessed from the target using the ASE BIOS services. These services allow communication between the target and the DA via the JTAG interface.

Host access

Channels are accessed from the host using the Debug Interface Adapter Library (DIAL). This library provides communication between the host and the DA via the SCSI interface.

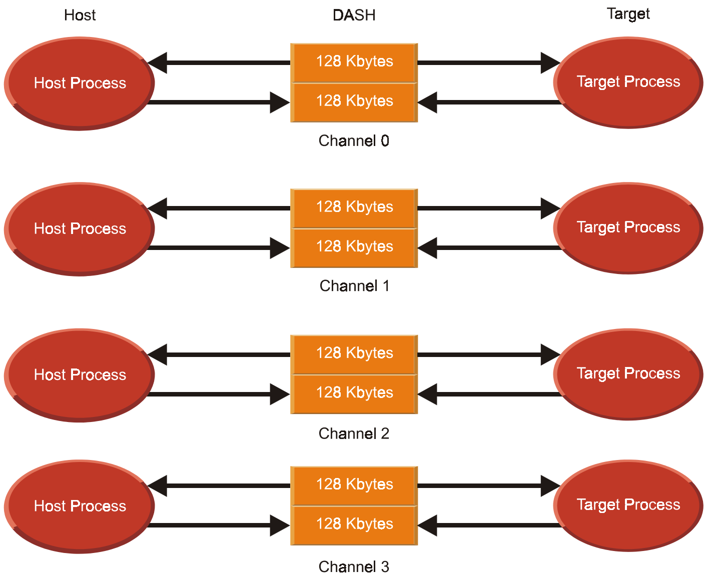
Channel designation

The channels are reserved for use as follows:

Channel Number	Use
0	Operating System debugging interface.
1	File Server (Debugging File Server Protocol).
2	Sound.
3	Spare.

Channel buffers

The DA contains eight 128kbyte buffers to ensure uninterrupted data transfer between the target and host. Each of the four channels has a host-to-target and a target-to-host buffer as shown:



ASE BIOS services

This section describes the ASE BIOS calls used to access channels from the target, channel interrupts and code examples.

Target channel access

Channel access from the target to the application is implemented using either:

- ASE BIOS calls made by the application software to the debug stub.

  -OR-

- Interrupts issued by the DA debug stub to the target application interrupt handler.

The ASE BIOS calls are non-blocking to allow the target application to issue a BIOS call and receive a reply immediately. The return values of the reply indicate the success or failure to carry out the requested task.

BIOS calls

Six ASE BIOS calls are available for use by the application software, these are:

- Read byte, INCHR.
- Write byte, OUTCHR.
- Read buffer, RDBF.
- Write buffer, WRBF.
- Read channel buffer status, RDSTAT.
- Set and read channel interrupts, CHISR.

Interrupts

The DA can issue an interrupt to the target application interrupt handler when a specified channel buffer condition is met, *see "Set and Read Channel Interrupts, CHISR" on page 29*.

Accessing ASE BIOS services

In privileged mode the ASE BIOS services are accessed by:

1. Loading registers R4, R5, R6, R7 with the required parameters.
2. JSR to the start address of the debug stub, ASE_BIOS in the examples.

In user mode the ASE BIOS services are accessed by:

1. Loading registers R4, R5, R6, R7 with parameters.
2. TRAPA #02.

Notes:

1. *ASE BIOS services are provided by the Extended debug stub resident in target system RAM. The ASE debug stub provides just one service, "Instruct the DA where to load the Extended stub".*
2. *The ASE BIOS accepts any value for the destination address, therefore calling code should validate the supplied parameters in the TRAPA handler.*
3. *If an application exception handler is in use, the debug stub exception passback must be implemented to allow the TRAPA #02 to be handled by the debug stub. The application exception handler must not alter the values of EXPEVT and TRA prior to issuing the passback call.*
4. *In privileged mode the TRAPA #02 instruction can also be used to issue a BIOS call subject to the same validation as above.*
5. *BIOS calls can be issued using TRAPA #02 when the debug stub's default exception handler is in use. In this instance the default handler will issue the BIOS call in a way that is transparent to the application code.*

Setup registers

To use the ASE BIOS services, set up the registers as shown in the following table. Request the service by a JSR to the start address of the debug stub, ASE_BIOS, if in privileged mode, or a TRAPA #02 if in user mode.

Name	Description	R4	R5	R6	R7	LOC
INCHR	Read byte	0x0A00 + Channel Number.	Not used.	Not used.	Not used.	Not used.
OUTCHR	Write byte.	0x0B00 + Channel Number.	Byte to write.	Not used.	Not used.	Not used.
RDBF	Read buffer.	0x0A08 + Channel Number.	Maximum write size.	Destination address.	Address of LOC.	Number of bytes read.
WRBF	Write buffer.	0x0B08 + Channel Number.	Number of bytes to write.	Source address.	Address of LOC.	Number of bytes written.
RDSTAT	Read channel buffer status.	0x0C00 + Channel Number.	Not used.	Not used.	Address of LOC.	Status.

Results

After the BIOS call, R0 and LOC hold the results as follows:

Operation	R0	LOC
INCHR	Byte read in low word, error code in high word.	Not used.
OUTCHR	Error code in high word.	Not used.
RDBF	Error code in high word.	Number of bytes read.
WRBF	Error code in high word.	Number of bytes written.
RDSTAT	Error code in high word.	Data available, space available.

Error codes

There are nine possible error codes, stored in the high order 16 bits of R0:

Name	Value	Meaning
CONAOK	0	No error, the operation was successful.
CONERR	1	Fatal error. The DA suffered an internal error.
CONBAD	2	Unknown command.
CONPRM	3	Parameter error. A buffer count of zero was requested.
CONADR	4	Bad address.
CONCNT	5	Bad count. A buffer count exceeding the maximum allowable value was requested. The current maximum value is 65536 bytes.
CONCBF	6	Channel buffer full. If there is insufficient space to write all the requested number of bytes, then as much data will be written as possible. When the buffer is full the application is informed that no further data was written because there was no more space available.
CONCBE	7	Channel buffer empty. The application is informed that there was no data to read.
CONBSY	8	Channel busy.

NOTE: *An ASE BIOS call may use a subset of the above codes for its return codes.*

ASE BIOS calls

Read byte, INCHR

Reads a single byte of data from the specified channel buffer to the target.

Call parameters

R4	0xA00 + channel (0,1,2,3)
R5	Not used.
R6	Not used.
R7	Not used.
LOC	Not used.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBE, channel buffer empty. CONERR, fatal error. Byte read in low order 8 bits.
LOC	Not used.

Example BIOS call

```
; OS (Channel 0) issue a 'read byte non-blocking' BIOS call.
   mov.l     #(INCHR + OS),r4 ;BIOS command + channel (0xA00 + 0).
   mov.l     #STUBSTART,r9    ;Point at the stub start.
   jsr       @r9              ;Issue BIOS call
   nop
; After call:
; Byte Read returned in low word R0.
; Error code returned in high word R0.
```

Write byte, OUTCHR

Writes a single byte of data from the target to the specified channel buffer.

Call parameters

R4	0xB00 + channel (0,1,2,3).
R5	Byte of data to write.
R6	Not used.
R7	Not used.
LOC	Not used.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBF, Channel buffer full. CONERR, Fatal error.
LOC	Not used.

Example BIOS call

```
; OS (Channel 0) issue a 'write byte non-blocking' BIOS call.
   mov.l  #(OUTCHR + OS),r4   ;BIOS command + channel (0xB00 + 0).
   mov.l  #0xa5,r5            ;Byte to write (e.g. 0xa5).
   mov.l  #STUBSTART,r9       ;Point at the stub start.
   jsr    @r9                 ;Issue BIOS call
   nop
; After call:
; Error code returned in high word R0.
```

Read buffer, RDBF

Reads a number of bytes of channel data, up to a specified maximum.

Call parameters

R4	0xA08 + channel (0,1,2,3).
R5	Byte Count. The maximum transfer size of data to be read from the specified channel buffer.
R6	Destination address of data read from the specified channel buffer.
R7	Zero, or address of LOC to put Byte Count.
LOC	Zero or a 32-bit address to store the actual number of bytes read from the specified channel buffer.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONPRM, Bad parameter. CONCNT, Bad count. CONCBE, Channel buffer empty. CONERR, Fatal error.
LOC	The number of bytes actually read (32-bits) or zero if the channel buffer was empty.

Example BIOS call

```
; OS (Channel 0) issue a 'read buffer non-blocking' BIOS call.
   mov.l  #(RDBF + OS),r4  ;BIOS command + channel (0xA08 + 0).
   mov.l  #200,r5          ;Max number of bytes to read.
   mov.l  #buffer,r6       ;Pointer to destination buffer area.
   mov.l  #loc,r7          ;Location to report actual number
                           ;of bytes read after BIOS call.
   mov.l  #STUBSTART,r9    ;Point at the stub start.
   jsr    @r9              ;Issue BIOS call
```

```
        nop
    ; After call:
    ; Error code returned in high word R8.
    ; 'Loc' contains the number of bytes read.
    ; 'buffer' contains 'loc' number of data bytes read.
```

Write buffer, WRBF

Writes a number of bytes, up to a specified maximum, to the specified channel buffer.

Call parameters

R4	0xB08 + channel (0,1,2,3).
R5	Byte Count. The number of bytes to be written to the specified channel buffer, up to a maximum of 65536 bytes.
R6	Source address. The address in target memory to read data from.
R7	Zero, or address of LOC to put the Byte Count. If this value is non-zero, then the value points to a 32-bit location to store the actual number of bytes written to the DA buffer.
LOC	32-bit location to hold the number of bytes transferred.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBF, Channel buffer full. CONERR, Fatal error.
LOC	Number of bytes actually written (32 bits) or zero if the specified channel buffer was full.

Example BIOS call

```
; OS (Channel 0) issue a 'write buffer non-blocking' BIOS call.
   mov.l  #(WRBF + OS),r4  ;BIOS command + channel (0xB08 + 0).
   mov.l  #200,r5          ;Max number of bytes to write.
   mov.l  #buffer,r6       ;Pointer to source buffer area.
   mov.l  #loc,r7          ;Location to report actual number
                                 ;of bytes written after BIOS call.
   mov.l  #STUBSTART,r9    ;Point at the stub start.
   jsr    @r9              ;Issue BIOS call
   nop
; After call:
; Error code returned in high word R8.
; 'Loc' contains the number of bytes written.
```

Read channel buffer status, RDSTAT

Informs the target application of amount of data available to be read from and space available to be written to in the specified channel buffer.

Call parameters

R4	0xC00 + channel (0,1,2,3).
R5	Not used.
R6	Not used.
R7	Address of LOC to buffer status information.
LOC	64-bit location to hold bytes of data available (long), bytes of space available (long).

Returned data

R0	CONAOK.
LOC	64-bit location to hold bytes of data available (long), followed by bytes of space available (long).

Example BIOS call

```
; OS (Channel 0) issue a 'read channel buffer status' BIOS call.
   mov.l  #(RDSTAT + OS),r4   ;BIOS command + channel (0xC00 + 0).
   mov.l  #loc,r7             ;Location to report the actual
                              ;status information
   mov.l  #STUBSTART,r9       ;Point at the stub start.
   jsr    @r9                 ;Issue BIOS call
   nop
; After call:
; Error code returned in high word R8.
; 'Loc' (long) contains the number of bytes available to be read.
; 'Loc'+4 (long) contains the space in bytes available for writing.
```

Set and Read Channel Interrupts, CHISR

Use CHISR to service buffers:

- Request a Hitachi-UDI (HUDI) interrupt if a transmitting buffer is empty or a receiving buffer is full.
- Determine if data is present in a channel buffer.
- Acknowledge receipt of HUDI interrupts by the application interrupt handler.

Call Parameters

R4	CHISR = 0x900
R5	Address of a long word LOC, must be non-zero.
R6	Not used.
R7	Not used.
LOC	Four 8-bit fields, as described below.

			Channel		
Name	0	1	2	3	Set bit to...
IPISET[1]	0 (LSB)	8	16	24	change input buffer interrupts (bits 1-2).
IPSIE[5]	1	9	17	25	enable a single input buffer interrupt, clear to disable. Valid only if bit 0 is set.
IPMIE[3]	2	10	18	26	enable multiple interrupts, clear to disable. Valid only if bit 0 is set.
IPACK	3	11	19	27	acknowledge receipt of interrupt when data is present in the input buffer.
OPISET[2]	4	12	20	28	change output buffer interrupts (bits 5-6).
OPSIE[5]	5	13	21	29	enable a single output buffer interrupt, clear to disable. Valid only if bit 4 is set.
OPMIE[4]	6	14	22	30	enable multiple interrupts, clear to disable. Valid only if bit 4 is set.
OPACK	7	15	23	31(MSB)	acknowledge receipt of an interrupt when space is available in the output buffer.

Returned data

R0	One of CONAOK, CONERR, CONPRM in the high order 16 bits.
LOC	Four 8-bit fields, as described below.

Name	Channel 0	1	2	3	Meaning when bit set
IPINT	0 (LSB)	8	16	24	An input buffer interrupt has occurred.
IPSIE	1	9	17	25	Input buffer interrupts are enabled.
IPMIE	2	10	18	26	Multiple interrupts are enabled.
IPRDY	3	11	19	27	There is data in the input buffer.
OPINT	4	12	20	28	An output buffer interrupt has occurred.
OPSIE	5	13	21	29	Output buffer interrupts are enabled.
OPMIE	6	14	22	30	Multiple interrupts are enabled.
OPRDY	7	15	23	31 (MSB)	There is space in the output buffer.

Bits 0-7 apply to channel 0, 8-15 to channel 1, 16-23 to channel 2, and 24-31 to channel 3.

Notes

1. *IPSIE and IPMIE are ignored if IPISET is set to 0.*
2. *OPSIE and OPMIE are ignored if OPISET is set to 0.*
3. *IPMIE is ignored if IPSIE is set to 0.*
4. *OPMIE is ignored if OPSIE is set to 0.*
5. *IPSIE and OPSIE are cleared automatically on acknowledgment of an interrupt.*

Servicing buffers

There are two ways to use the CHISR service to inform the application software there is a buffer for it to service:

- Interrupt control. Trigger a HUDI interrupt when channel buffer data becomes available or channel space is available.
- Polling. Poll a channel to determine if a transmitting buffer is empty or a receiving buffer is non-empty.

Interrupt control

The CHISR BIOS call allows the target application to interrupt the target if one of these conditions are met:

- Data is in the input buffer for the application software to read and process.
- Space is available in the output buffer for the application software to store data in the buffer, ready for the host to read and process.

You can generate single or multiple HUDI interrupts when a condition is met. Single interrupts are generated once-only when an interrupt condition is met. Multiple interrupts are generated each time the channel condition is met.

Notes:

1. *The HUDI is not an ASE mode interrupt.*
2. *The HUDI interrupt vectors to the application software interrupt handler. It is the responsibility of the application software to ensure this is present.*
3. *CHISR can return with more than one 'interrupt occurred' bit set.*

Example BIOS call

This example shows a typical BIOS call to set up and clear interrupts for the four channels.

```
; Use the 'set and read interrupts' BIOS call to set up interrupts.
;
   mov.l  #loc,r5      ;Location for set up parameters.
;
; Channel 0. Operating system. Set for single interrupt on input
; buffer (data available).
   mov.b  #(IPISET + IPSIE),r8
   mov.b  r8,@r5
;
; Channel 1. File server. Set up for multiple interrupts on input
; buffer (data available) and output buffer (space available).
    mov.b #(IPSET + IPMIE + OPISET + OPMIE),r8
    mov.b r8,@(1,r5)
;
; Channel 2. Sound tools. Turn off interrupts.
   mov.b  #(IPISET + OPISET),r8
   mov.b  r8,@(2,r5)
;
; Channel 3. Unused. Make no changes to channel 3.
   mov.b  #0x00,r8
   mov.b  r8,@(3,r5)
;
; Issue a 'Set and read channel interrupts' BIOS call.
   mov.l  #CHISR,r4        ;BIOS command 0x900.
   mov.l  #STUBSTART,r9    ;Point at the stub start.
   jsr    @r9              ;Issue BIOS call.
   nop
; After call:
; Error code returned in high word R8.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
; status.
```

Polling

The CHISR, ‘Set and Read Channel Interrupts', BIOS call can be used to determine the state of individual channel buffers. This allows polling to implemented instead of interrupt control.

Example BIOS call

This example shows a typical BIOS call to interrogate the four channels' buffers.

```
; Use the 'set and read interrupts' BIOS call to read channel status.
   mov.l  #loc,r5      ;Location for set up parameters.
   mov.b  #0,r8
   mov.b  r8,@rr5          ;No change channel 0
   mov.b  r8,@(1,r5)       ;No change channel 1
   mov.b  r8,@(2,5)        ;No change channel 2
   mov.b  r8,@(3,r5)       ;No change channel 3
;
; Issue a 'Set and read channel interrupts' BIOS call.
   mov.l  #CHISR,r4        ;BIOS command 0x900.
   mov.l  #STUBSTART,r9    ;Point at the stub start.
   jsr    @r9              ;Issue BIOS call.
   nop
; After call:
; Error code returned in high word R8.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
; status.
   mov.l  #loc,r5      ;Location for returned parameters.
   mov.b  @r5+,r1      ;Current Channel 1 status.
   mov.b  @r5+,r2      ;Current Channel 2 status.
   mov.b  @r5+,r3      ;Current Channel 3 status.
   mov.b  @r5+,r4      ;Current Channel 4 status.
; The registers r1,r2,r3,r4 now contain the details for each channel
; for the following items:
;  Input buffer data available/no data available.
;  Output buffer space available/no space available.
; and also:
;  Input buffer occurred/not occurred.
;  Input buffer interrupts enabled/disabled.
;  Input buffer multiple interrupts enabled/disabled.
;  Output buffer occurred/not occurred.
;  Output buffer interrupts enabled/disabled.
;  Output buffer multiple interrupts enabled/disabled.
```

Acknowledging receipt of a HUDI interrupt

You can use CHISR to acknowledge the receipt of an HUDI interrupt. The application interrupt handler must include the functionality to acknowledge HUDI interrupts for the following reasons:

1. The HUDI interrupt cannot be guaranteed to be accepted by the SH4. This is a characteristic of the SH4 microprocessor.
2. The interrupt can be lost if the debug stub is entered at the same time as a HUDI interrupt is generated, for example when executing a software BRK breakpoint.

The DA will repeatedly issue HUDI interrupts (when interrupts are enabled) until the application software has acknowledged the interrupt by calling CHISR.

When the application software code HUDI interrupt handler is entered due to an HUDI interrupt, the 'set and read channel interrupts' command with the relevant acknowledge bits set should be issued. This will inform the DA that the interrupt has been accepted and prevent it from re-issuing the HUDI interrupt again.

Example BIOS call

This example shows a typical BIOS call to acknowledge an input interrupt on channel 1.

```
; Use the 'set and read interrupts' BIOS call to acknowledge an
interrupt.
   mov.l  #loc,r5          ;Location for set up parameters.
   mov.b  #0,r8
   mov.b  r8,@r5           ;No change channel 0
   mov.b  r8,@(2,r5)       ;No change channel 2
   mov.b  r8,@(3,r5)       ;No change channel 3
   mov.b  #IPACK,r8        ;Acknowledge interrupt.
   mov.b  r8,@(1,r5)
;
; Issue a 'Set and read channel interrupts' BIOS call.
   mov.l  #CHISR,r4        ;BIOS command 0x900.
   mov.l  #STUBSTART,r9    ;Point at the stub start.
   jsr    @r9              ;Issue BIOS call.
   nop
; After call:
; Error code returned in high word R8.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
; status.
```

HUDI interrupt handler considerations

HUDI interrupt handler

The interrupt associated with channel usage generated by the DA is the HUDI interrupt. This causes an SH4 'interrupt' category at an offset of 0x600 from the VBR. The application code must have a suitable exception handler routine in place to allow the exception to be serviced.

As with the SH4's entire on-chip peripheral module interrupts the HUDI interrupt has an associated priority to allow masking of interrupts by using the status register interrupt level mask bits (SR.IMASK). This priority must be set so that the HUDI interrupt is not masked. The register IPRC (interrupt priority register C) facilitates the setting up of the priority for the HUDI. It is the responsibility of the application program to ensure that the priority for the HUDI is high enough to allow interrupts through. It is recommended that the IPRC value be set to its highest level (15) to ensure unplanned masking of this interrupt does not occur.

It must be noted that the on-chip peripheral module interrupt priority level setting should only be performed when the status register block bit SR.BL is set. This is a requirement of the SH4.

Within the application code interrupt handler, the INTEVT register must be checked to ascertain the source of the interrupt. For the HUDI interrupt the INTEVT will contain 0x6000x5E0 and EXPEVT will contain 0x620.

BIOS calls within the HUDI interrupt handler

The recommended method for implementing a HUDI interrupt handler is:

1. Issue a CHISR (Set and Read Channel Interrupts) BIOS call to ascertain which channel sources caused the interrupt (and whether it was on input and/or output).
2. For each of the interrupting sources, either:
   - Handle the interrupt immediately by issuing further BIOS channel commands to move data between the Host and the target as applicable.

     -OR-
   - Set flags or signals to inform 'background code' of the interrupt occurring.

   Issue a CHISR (Set and Read Channel Interrupts) BIOS call to acknowledge all channel sources that caused the interrupt.

NOTE: *The above method assumes that all the normal requirements for the implementation of SH4 interrupt handlers are observed.*

CPDIAL Library Reference

Host channel access

The CPDIAL communications library provides access to channels from the host. CPDIAL is organized as a C++ class called *CDial* containing objects that each hold an API for a different device supported by the library. There are four header files required to use the library with channels, the principle header file is *dial.h*.

Header file:	Contains:
Dial.h	The *CDial* class definition and definition of *DIALDEVICE*.
DialChannel.h	The definition of *CDialChannel*, the base class for the six channel-specific classes. Data structures used by *CDIALChannel* are contained in *GenChannel.h*.
GenChannel.h	The data structures used by *DialChannel*.
error.h	The DIAL API error code definitions.

Connecting and disconnecting

Two functions, *Connect* and *Disconnect* establish and release a connection between an application and a device. Depending on the type of the device, the nature of the connection can be very important. For example, a network device may be accessed by multiple users but an active connection provides a user with exclusive access to the device for the duration of that connection.

NOTE: *When a device is first found (using* `FindNextDevice`*) it is recommended that a connection is made and kept until the device is no longer required. For example, CodeScape connects to a device, locally or across a network, at startup and does not disconnect from the device until it is shut down.*

Using DIAL

To initialize and use the library acquire a pointer to the library using *InitializeDial* function.

Calling a function

DIAL requires a ‘magic cookie’ (denoted by *DIAL_ID*) that the DIAL DLL uses to identify the device. The *DIAL_ID* is usually obtained using the *FindNextDevice* function.

Example

NOTE: *This example illustrates the calling convention only; for more examples, see the TESTDIAL diagnostic utility and source code provided with CPDIAL.*

```
// Instantiate the DIAL library once only for any given application
CDial * pDial = InitializeDial();
pDial = GetDial;

DIAL_EC ecRetCode = DIAL_EC_NOERROR;
DIAL_ID DeviceID = DIAL_ID_UNDEF;

ecRetCode = DIAL->FindNextDevice(     DeviceID,
                                      DIALDEVICE::DEVTYPE_KATANA_DA);
// a DIAL root method

if (ecRetCode == DIAL_EC_NOERROR)
{
  ecRetCode = pDial->DA.xxx(DeviceID, ...)  // control the DA
  ecRetCode = pDIAL->Console.xxx(DIAL_ID, deviceID, ...) // control the console.
  ...                                                     // access the channels.
  ecRetCode = DIAL->FServer.Reserve (DeviceID, ...);
  ecRetCode = DIAL->FServer.Read(DeviceID, ...);
  ecRetCode = DIAL->FServer.Release(DeviceID, ...);
}
```

CDial functions

The *CDial* class contains these functions:

Function name	Purpose
InitializeDial	Initializes DIAL and returns a pointer to the Dial interface.
GetVersion	Returns the library's version number.
SetTimeOut	Sets the timeout period.
GetTimeOut	Returns the current timeout value.
FindNextDevice	Locate devices on the bus.
GetDeviceDetails	Supplies details of a specified device.
ValidateDevice	Tests the validity of a specified device.
GetErrorText	Converts an error code to a strings.
Connect	Makes a connection to a device.
Disconnect	Disconnects from a device.

The following two functions are available for the Dreamcast:

GetDialDebugMode	Gets the current debug mode, OS or CPU.
SetDialDebugMode	Sets the current debug mode, OS or CPU.

Error codes

The majority of *CDial* class methods return a 32-bit error code (of type *DIAL_EC*). The higher 16 bits is a group code indicating the module, the lower 16 bits is an error code indicating the error. Error codes are defined in the file *error.h*; the construct of the error is not usually required to be known. Passing the error code to the *GetErrorText* function returns a textual description of the category and specific error encountered.

CDial class definition

The *CDial* class definition has the public form. The class *CChannel* is defined externally to *CDial* in its own definition and implementation files and contains just the routines specific to that module.

```
class CDial
{
public:
               enum DEBUGMODE
               {
                   DEBUGMODE_OS,
                   DEBUGMODE_CPU
               };
               CDialDAEx&        DA;
               CDialConsoleEx&   Console;

               // Use these channel wrapper classes in preference to using
               // CChannel directly
               CTypedChannelEx&      VSerial;
               CTypedChannelEx&      DebugOS;
               CTypedChannelEx&      FServer;
               CTypedChannelEx&      Sound;
               CTypedChannelROEx&    TraceProfile;
               // Direct access to the CChannel class.
               // Needs specification of channel identifier CHANTYPE enumerated
               // parameter defined in channel.h
               // N.B. CHANTYPE_TRACEPROFILE is *Read Only*, as specified in
               // CTraceProfile API.
               CDialChannelEx&   Channel;

               CDial( CDialDAEx& da,
                       CDialConsoleEx& console,
                       CTypedChannelEx& vserial,
                       CTypedChannelEx& debugOS,
                       CTypedChannelEx& fileServer,
                       CTypedChannelEx& sound,
                       CTypedChannelROEx& traceProfile,
                       CDialChannelEx& channel
                       ) : DA( da ),
                           Console( console ),
                           VSerial( vserial ),
                           DebugOS( debugOS ),
                           FServer( fileServer ),
                           Sound( sound ),
                           TraceProfile( traceProfile ),
                           Channel( channel )
               {
               }
               virtual BOOL      GetVersion( DWORD&, DWORD& ) const = 0;

               virtual void      SetTimeOut( DWORD ) = 0;
               virtual DWORD     GetTimeOut() const = 0;

               virtual DIAL_EC   FindNextDevice( DIAL_ID&,
                                                DIALDEVICE::DEVTYPE ) = 0;

               virtual DIAL_EC   GetDeviceDetails( DIAL_ID, DIALDEVICE& ) = 0;

               virtual DIAL_EC   ValidateDevice( DIAL_ID,
                              DIALDEVICE::DEVTYPE = DIALDEVICE::DEVTYPE_UNDEF ) =
0;

               virtual const char*GetErrorText( DIAL_EC ) = 0;

               virtual DIAL_EC   Connect( DIAL_ID ) = 0;

               virtual void      Disconnect( DIAL_ID ) = 0;
};

#ifdef DIAL_DEF_
extern "C" __declspec( dllexport ) CDial * InitializeDial();
extern "C" __declspec( dllexport ) DIAL_EC GetDialDebugMode(CDial *,
                                                            DIAL_ID,
                                                            CDial::DEBUGMODE& );
extern "C" __declspec( dllexport ) DIAL_EC SetDialDebugMode(CDial *,
                                                            DIAL_ID,
                                                            CDial::DEBUGMODE );
#else
extern "C" __declspec( dllimport ) CDial * InitializeDial();
extern "C" __declspec( dllimport ) DIAL_EC GetDialDebugMode(CDial *,
                                                            DIAL_ID,
                                                            CDial::DEBUGMODE& );
extern "C" __declspec( dllimport ) DIAL_EC SetDialDebugMode(CDial *,
                                                            DIAL_ID,
                                                            CDial::DEBUGMODE );
typedef    CDialDAEx          CDA;
typedef    CDialConsoleEx     CConsole;
typedef    CDialChannelEx     CChannel;
typedef    CDialMirageEx      CMirage;
#endif
```

InitializeDial

Initializes DIAL and returns a pointer to the Dial interface.

```
CDial * InitializeDial();
```

Return value

A pointer to the Dial interface.

Parameters

None.

GetVersion

Returns the library’s version number.

```
BOOL CDial::GetVersion (   WORD& Major,
                           WORD& Minor);
```

Return value

Always returns *True*.

Parameters

Major
: The major version number.

Minor
: The minor version number.

SetTimeOut

Sets the timeout, in milliseconds, for the high-level commands.

```
VOID SetTimeOut(DWORD timeout);
```

Return value

None.

Parameters

timeout
The timeout value in milliseconds, the default is 5000 milliseconds.

Remarks

Sets the timeout value for the channel-specific commands. If a command reports a non-serious error, such as *BUSY* or *PENDING*, DIAL retries the command or waits for it to be completed until the timeout period is reached.

GetTimeOut

Returns the current timeout value for DIAL high-level commands.

```
DWORD GetTimeOut(VOID);
```

Return value

The current timeout value in milliseconds.

Parameters

None.

FindNextDevice

Locate devices on the bus.

```
DIAL_EC FindNextDevice(   DIAL_ID& device,
                          DIALDEVICE::DEVTYPE type = DIALDEVICE::DEVTYPE_INDEF);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: The most recently found device. Use *DIAL_ID_UNDEF* to find the first device.

type
: The type of device to find. The default value is *ALL*. *See DIALDEVICE* in *dial.h* for a list of device types.

Remarks

Use *FindNextDevice* to find DIAL devices. The caller supplies a device identifier (or *DIAL_ID_UNDEF* to find the first device). On successful return the device identifier of the next matching device is set.

NOTE: *Device identifiers are passed by reference and will be modified.*

GetDeviceDetails

Supplies details for a specified device.

```
DIAL_EC GetDeviceDetails(     DIAL_ID device,
                              DIALDEVICE& details);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifer.

details
: A *DIALDEVICE* structure where the results are stored.

Remarks

See DIALDEVICE in *dial.h* for supported device types.

NOTE: *The details of a particular device are passed by reference and will be modified.*

ValidateDevice

Tests the validity of a specified device.

```
DIAL_EC ValidateDevice(   DIAL_ID device,
                          DIALDEVICE::DEVTYPE type = DIALDEVICE::DEVTYPE_UNDEF);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
A valid device identifer.

type
The type of device to validate. The default is *DIALDEVCE::DEVTYPE_UNDEF*.

Remarks

See DIALDEVICE in *dial.h* for a list of device types.

GetErrorText

Converts an error code to a string for display purposes.

```
const char * GetErrorText(DIAL_EC error)
```

Return value

Returns a pointer to a string containing a textual representation of *error*.

Parameters

error
: The error code to convert.

Remarks

None.

Connect

Makes a connection to a device.

```
DIAL_EC Connect(DIAL_ID device)
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

The identifier of the device to connect to.

Remarks

The device will not be available to other users until it is released using the *Disconnect* function. Each use of *Connect* must be matched by a corresponding *Disconnect* call.

NOTE: Always use the Connect *and* Disconnect *functions connecting to a device. This is the most efficient method and guarantees the correct state of the device.*

Disconnect

Disconnects from a device.

```
void Disconnect(DIAL_ID device)
```

Return value

None.

Parameters

device
: The identifier of the device to disconnect from.

Remarks

After disconnecting, the device is then available to other users. Each use of the *Connect* function must be matched by a corresponding *Disconnect* call.

NOTE: *Always use the* Connect *and* Disconnect *functions connecting to a device. This is the most efficient method and guarantees the correct state of the device.*

GetDialDebugMode

Gets the current debug mode (either *CDial::DEBUGMODE_OS* or *CDial::DEBUGMODE_CPU*).

```
DIAL_EC GetDialDebugMode(  CDial * pDial,
                           DIAL_ID device,
                           CDial::DEBUGMODE& mode );
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

pDial
: Pointer to DIAL (as returned by *InitializeDial*).

device
: A valid device identifer.

mode
: Storage for the result.

Remarks

None.

SetDialDebugMode

Sets the current debug mode (either *CDial::DEBUGMODE_OS* or *CDial::DEBUGMODE_CPU*).

```
DIAL_EC SetDialDebugMode(  CDial * pDial,
                           DIAL_ID device,
                           CDial::DEBUGMODE mode );
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

pDial
: Pointer to DIAL (as returned by *InitializeDial*).

device
: A valid device identifer.

mode
: The mode to set.

Remarks

None.

CDial::CDialChannelEx functions

Typed channels

DIAL's channel functions are defined by three classes, *CDialChannelEx*, *CTypedChannelROEx* and *CTypedChannelEX*. *CDialChannelEx* represents a generic channel whereas *CTypedChannelROEx* and *CTypedChannelEX* represent channels of a particular type.

CDialChannelEx class definition

```
class CDialChannelEx : public CGenChannel
{
public:
               virtual DIAL_EC Reserve(   DIAL_ID,
                                          CHANTYPE,
                                          DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
               virtual DIAL_EC Release( DIAL_ID, CHANTYPE ) = 0;
               virtual DIAL_EC Validate( DIAL_ID, CHANTYPE ) = 0;
               virtual DIAL_EC DataReady( DIAL_ID, CHANTYPE, BOOL&, DWORD& ) = 0;
               virtual DIAL_EC Read(  DIAL_ID,
                                      CHANTYPE,
                                      DWORD,
                                      DWORD&,
                                      void*,
                                      BOOL = FALSE,
                                      DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
               virtual DIAL_EC Write( DIAL_ID,
                                      CHANTYPE,
                                      DWORD,
                                      DWORD&,
                                      const void*,
                                      BOOL = FALSE,
                                      DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```

Reserve

Reserves the channel specified by *type* and must be used before any reading or writing is performed.

```
DIAL_EC Reserve(       DIAL_ID device,
                       CHANTYPE type,
                       DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

type
: A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

timeout
: The length of time to wait for the channel if it is in use by another process.

Remarks

When the channel is no longer required, for example when a program exits, a corresponding Release call must be made.

If the channel is currently in use, the length of time *Reserve* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

Release

Releases the channel specified by *type* on the device specified by *device* for use by other processes.

```
DIAL_EC Release(      DIAL_ID device,
                      CHANTYPE type);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

type
: A member of the *CChannel::CHANTYPE* enumeration specifying the channel.

Validate

Tests for the presence of the channel specified by *type* on the device specified by *device* and is reserved for use by this process.

```
DIAL_EC Validate(      DIAL_ID device,
                       CHANTYPE type);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

type
: A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

Remarks

None.

DataReady

Determines whether there is any data to read on the specified channel.

```
DIAL_EC DataReady(     DIAL_ID device,
                       CHANTYPE type,
                       BOOL& isData,
                       DWORD& status);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

type
: A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

isData
: Set to *True* if data is available in the buffer, *False* otherwise.

status
: Holds additional status information on return.

Remarks

Currently only the *CHANTYPE_TRACEPROFILE* channel modifies this value to *CHANSTATUS_HALFFULL*. For all other specified channels, it is returned as 0.

Read

Reads a specified amount from the specified channel and returns the actual amount read.

```
DIAL_EC Read(      DIAL_ID device,
                   CHANTYPE type,
                   DWORD size,
                   DWORD& sizeRead,
                   const void* buffer,
                   BOOL blocking = FALSE,
                   DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

type
: A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

size
: The amount of data to read.

sizeRead
: The amount of data actually read.

buffer
: A pointer to a buffer to store the result.

blocking
: Set to *True* for blocking, *False* for non-blocking.

timeout
: The delay before a timeout.

Remarks

By default *Read* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Read* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify a value of *INFINITE*.

Write

Writes from the specified channel to the specified address and returns the actual number of bytes read.

```
DIAL_EC Write(     DIAL_ID device,
                   CHANTYPE type,
                   DWORD size,
                   DWORD& sizeWritten,
                   const void* buffer,
                   BOOL blocking = FALSE,
                   DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
A valid device identifier.

type
A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

size
The amount of data to read.

sizeWritten
The amount of data actually read.

buffer
A pointer to a buffer to store the result.

blocking
Set to *True* for blocking, *False* for non-blocking.

timeout
The delay before a timeout.

Remarks

By default *Write* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Write* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify a value of *INFINITE*.

CDial::CTypedChannelROEx functions

CTypedChannelROEx class definition

```
//          This represents a read-only channel of a specific type

class CTypedChannelROEx : public CGenChannel
{
public:
               virtual DIAL_EC Reserve(   DIAL_ID,
                                          DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
               virtual DIAL_EC Release(   DIAL_ID ) = 0;
               virtual DIAL_EC Validate(  DIAL_ID ) = 0;
               virtual DIAL_EC DataReady(     DIAL_ID,
                                              BOOL&,
                                              DWORD& ) = 0;
               virtual DIAL_EC Read(      DIAL_ID,
                                          DWORD,
                                          DWORD&,
                                          void*,
                                          BOOL = FALSE,
                                          DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```

Reserve

Reserves the channel specified by *type* and must be used before any reading or writing is performed.

```
DIAL_EC Reserve(       DIAL_ID device,
                       DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

timeout
: The length of time to wait for the channel if it is in use by another process.

Remarks

When the channel is no longer required, for example when a program exits, a corresponding Release call must be made.

If the channel is currently in use, the length of time *Reserve* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

Release

Releases the channel specified by `type` on the device specified by *device* for use by other processes.

```
DIAL_EC Release(       DIAL_ID device);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
A valid device identifier.

Validate

Tests for the presence of the channel specified by *type* on the device specified by *device* and is reserved for use by this process.

```
DIAL_EC Validate(      DIAL_ID device);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

None.

DataReady

Determines whether there is any data to read on the specified channel.

```
DIAL_EC DataReady(     DIAL_ID device,
                       BOOL& isData,
                       DWORD& status);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

isData
: Set to *True* if data is available in the buffer, *False* otherwise.

status
: Holds additional status information on return.

Remarks

None.

Read

Reads a specified amount from the specified channel and returns the actual amount read.

```
DIAL_EC Read(      DIAL_ID device,
                   DWORD size,
                   DWORD& sizeRead,
                   const void* buffer,
                   BOOL blocking = FALSE,
                   DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

size
: The amount of data to read.

sizeRead
: The amount of data actually read.

buffer
: A pointer to a buffer to store the result.

blocking
: Set to *True* for blocking, *False* for non-blocking.

timeout
: The delay before a timeout.

Remarks

By default *Read* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time `Read` waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

CDial::CTypedChannelEx functions

CTypedChannelEx class definition

CTypedChannelEx inherits from *CTypedChannelROEx* and adds the *Write* function.

```
//          This represents a read-write channel of a specific type

class CTypedChannelEx : public CTypedChannelROEx
{
public:
               virtual DIAL_EC Write(    DIAL_ID,
                                         DWORD,
                                         DWORD&,
                                         const void*,
                                         BOOL = FALSE,
                                         DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```

Write

Writes from the specified channel to the specified address and returns the actual number of bytes read.

```
DIAL_EC Write(     DIAL_ID device,
                   DWORD size,
                   DWORD& sizeWritten,
                   const void* buffer,
                   BOOL blocking = FALSE,
                   DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device
: A valid device identifier.

size
: The amount of data to read.

sizeWritten
: The amount of data actually read.

buffer
: A pointer to a buffer to store the result.

blocking
: Set to *True* for blocking, *False* for non-blocking.

timeout
: The delay before a timeout.

Remarks

By default *Write* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Write* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

CDial::Console functions

CDial::Console class definition

These functions are used to communicate with the target attached to the DA and are defined in `DialConsole.h`.

```
class CDialConsole : public CGenConsole
{
public:
           virtual DIAL_EC Inquiry(   DIAL_ID,
                                      INQUIRY& ) = 0;
           virtual DIAL_EC ProcessInquiryError(  const INQUIRY&,
                                                 BOOL = TRUE,
                                                 BOOL = FALSE ) = 0;
           virtual DIAL_EC Execute( DIAL_ID, DWORD ) = 0;
           virtual DIAL_EC Suspend( DIAL_ID ) = 0;
           virtual DIAL_EC Resume( DIAL_ID ) = 0;
           virtual DIAL_EC ReadMemory(   DIAL_ID,
                                         DWORD,
                                         ELEMENTSIZE,
                                         DWORD,
                                         void* ) = 0;
           virtual DIAL_EC WriteMemory(  DIAL_ID,
                                         DWORD,
                                         ELEMENTSIZE,
                                         DWORD,
                                         const void* ) = 0;
           virtual DIAL_EC ReadContext(  DIAL_ID,
                                         CONTEXTMODE,
                                         WORD,
                                         void* ) = 0;
           virtual DIAL_EC WriteContext( DIAL_ID,
                                         CONTEXTMODE,
                                         WORD,
                                         const void* ) = 0;
           virtual DIAL_EC ReadConfig(   DIAL_ID,
                                         READCONFIG& ) = 0;
           virtual DIAL_EC ResetNoDebug( DIAL_ID ) = 0;
           virtual DIAL_EC ResetAndDebug( DIAL_ID ) = 0;
           virtual DIAL_EC MakeSafe( DIAL_ID ) = 0;
           virtual DIAL_EC GetValidConsoleStatus(   DIAL_ID,
                                                    CDialConsoleStatus *& ) = 0;
};

#ifndef CPL_INTERNAL
class CDialConsoleEx : public CDialConsole {};
#endif
```

Inquiry

Returns the current state of the target.

```
DIAL_EC Inquiry(       DIAL_ID device,
                       INQUIRY& inquiry);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

inquiry
: A `CConsole::INQUIRY` structure to store the result.

Remarks

The status information returned is state dependent. Within the `CConsole::INQUIRY` class returned, three fields are used to identify the current execution state of the target, the signal category and a signal identifier. The fields relate to enumerations within the `CConsole` class such as `CConsole::SIGNAL`, `CConsole::SIGCAT`.

Signal state	Execution state
`SIGNAL_RUNNING`	Executing application code.
`SIGNAL_STOPPED`	In debug stub as result of a `Suspend()` target command.
`SIGNAL_HALTED`	In debug stub as result of CPU exception *or* target command issued. Check `bySigCat` and `wSigID` to differentiate.
`SIGNAL_RESET`	Initial condition after `CConsole::ResetAndDebug()` or manual reset or power cycle causes the stub to (re)load when the DA is in CPU debug mode.
`SIGNAL_SAFE`	Indicates a `CConsole::MakeSafe()` command was issued.
`SIGNAL_FAIL`	Failed to load/reset the debug stub.
`SIGNAL_NODEBUG`	Running without target debug facilities.

NOTE: *The* `CConsole::Inquiry()` *command does not in itself cause a* `SIGNAL_HALTED` *exception to be reported, even though it is described as a* `CConsole` *command. The command is able to non-intrusively report the current status.*

When a signal value of either `SIGNAL_STOPPED`, `SIGNAL_HALTED`, `SIGNAL_RESET` or `SIGNAL_SAFE` is flagged, the current PC field is also available and valid.

When a signal value of `SIGNAL_HALTED` is flagged, the signal category and signal ID fields provide processor specific information for the exception or other reason for being in the debug stub.

Possible values of the signal category where `Inquiry` returns `SIGNAL_HALTED`.

Signal category	Means
`SIGCAT_UNDEFINED`	Reserved for future use.
`SIGCAT_ASE`	Currently defined for use on the 7091-EVA.
`SIGCAT_GENERAL`	General exception.
`SIGCAT_INTERRUPT`	Interrupt.
`SIGCAT_CHNLBLOCK`	Blocked channel (such as Fserver) operation in progress.
`SIGCAT_REPORTEVENT`	Check SIGID for event being reported.
`SIGCAT_PROFILEBUFF`	Statistical sampling and trace profiling buffer signal.

The `SIGCAT_REPORTEVENT` is currently only used to report that a CPU manual reset has been detected. The signal ID will be `SIGID_MANUALRESET`.

The `SIGCAT_CHNLBLOCK` category can be treated in the same way as `SIGNAL_RUNNING`, and target commands may be issued as normal. However, should the `CConsole::Resume` command be issued, the blocked serial channel operation will return False if it has not yet completed. The response to this signal category is to wait for the `CHNLBLOCK` signal category to be cleared. For example, `CConsole::Inquiry` will continue to be issued until the signal changes and a signal value such as `SIGNAL_RUNNING` is detected.

The signal ID value is specific to the CPU and matches the manufacturer's exception code for the given category. In general, the 'vector base' and 'offset' are encoded into 'category', then 'exception code' is the category specific ID.

To determine the reason for halting the target execution and entering the debug stub, the `SIGNAL_HALTED` state and signal category `SIGCAT_ASE` are both specified. The two causes of this are:

- A `CConsole` command was issued by the host.
- A software breakpoint (using ASE BRK) or hardware breakpoint (using ASE HBC) was encountered.

The signal identifier allows you to differentiate between ASE exceptions:

```
if (InquirySignal == CConsole::SIGNAL_HALTED)
```

```
{
  switch (InquirySignalCategory)
  {
    case CConsole::SIGCAT_ASE:
      switch (InquirySignalID)
      {
       case 0x04: // ASE-HBC BREAK
          // HBC breakpoint encountered break;
        case 0x08: // ASE-SW BREAK
          // BRK software breakpoint encountered break;
        case 0x10: // ASE-PIN BREAK
          // ASE Pin break encountered as a result of CConsolecommand being issued
          // (Inquiry() command does not cause ASE-PIN BREAK entry into the stub).
          // This means target was executing game code and would have reported
          // a SIGNAL_RUNNING if we hadn't issued a command to it. After all debug
         // commands have been issued, a CConsole::Resume() command should be issued
         // to exit the debug stub and continue normal execution
         break;

       default:
         break;
      }

      break;
    ...
    ...
    default:
    ...
    break;
  }
}
```

NOTE: *The* `CConsole::Inquiry()` *command does not in itself cause a* `SIGNAL_HALTED` *and* `ASE-PIN BREAK` *exception to be reported, even though it is described as a* `CConsole` *command; it is able to non-intrusively report the current status.*

ProcessInquiryError

Queries the `inquiry` data bit-fields retrieved from the `Inquiry` command.

```
CDail::ProcessInquiryError(       const INQUIRY& inquiry,
                                  BOOL needDebug = TRUE,
                                  BOOL reserved = FALSE);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

inquiry
: A `CConsole::INQUIRY` structure obtained using the Inquiry function.

needDebug
: Set to True if debug support is required, False otherwise.

reserved
: Reserved. Always set to False.

Remarks

Use this function to validate whether the target is in a state to support debugging using the `CConsole` commands for example, all stubs have been successfully loaded with debug support available. Specify the required level of functionality using the `NeedDebug` boolean parameter.

Execute

Causes execution to start on the target from the given address.

```
DIAL_EC Execute(      DIAL_ID device,
                      DWORD address);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address
    The address to start executing from.

Remarks

This function is equivalent to modifying the PC register using read and write context and then issuing a resume command.

After a `CConsole::Execute()` command has been successfully issued, the `CConsole::Inquiry()` command reports the signal status as `SIGNAL_RUNNING` unless an exception is encountered prior to completion. In this case the relevant signal, category and code is reported.

Suspend

This causes the target specified by `device` to suspend execution and enter the debug stub.

```
DIAL_EC Suspend(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_STOPPED` after a `CConsole::Suspend()` command has been successfully issued.

Resume

Causes execution on the target to resume from the point where it stopped.

```
DIAL_EC Resume(DIAL_ID deviceID);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

Remarks

Use this function to resume execution when the target has been stopped due to a suspend command, exception or general host command interaction such as the current PC. After a `CConsole::Resume()` command has been successfully issued, the `CConsole::Inquiry()` command reports the signal status as `SIGNAL_RUNNING`, unless an exception is encountered prior to completion. In this case the relevant signal, category and code will be reported.

ReadMemory

Reads the target's memory.

```
DIAL_EC ReadMemory(        DIAL_ID device,
                           DWORD address,
                           ELEMENTSIZE elementSize,
                           DWORD elementCount,
                           void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
    A valid device identifier.

address
    The address to read from.

elementSize
    A member of the `CConsole::ELEMENTSIZE` enumeration (`ELEMENTSIZE_BYTE`, `ELEMENTSIZE_WORD`, or `ELEMENTSIZE_LONG`) specifying the size of each element.

elementCount
    The number of elements to read.

buffer
    A pointer to a buffer large enough to hold the resulting data.

Remarks

NOTE: *The memory is retrieved into the specified destination buffer as a byte stream in the same order as in the target's memory, regardless of the* `elementSize` *specified.*

WriteMemory

Writes to the target's memory.

```
DIAL_EC WriteMemory(      DIAL_ID device,
                          DWORD address,
                          ELEMENTSIZE elementSize,
                          DWORD elementCount,
                          const void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

address
: The address to write to.

elementSize
: A member of the `CConsole::ELEMENTSIZE` enumeration (`ELEMENTSIZE_BYTE`, `ELEMENTSIZE_WORD`, or `ELEMENTSIZE_LONG`) specifying the size of each element.

elementCount
: The number of elements to write.

buffer
: A pointer to a buffer holding the data to be written.

NOTE: *The memory is written from the specified source buffer as a byte stream as ordered in the target's memory, regardless of the* `ElementSize` *specified.*

ReadContext

Reads a context from the target.

```
DIAL_EC ReadContext(      DIAL_ID device,
                          CONTEXTMODE mode,
                          WORD length,
                          void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

mode
: A member of the `CConsole::CONTEXTMODE` enumeration that specifies which set of internal registers to read.

length
: The size of the buffer to store the result.

buffer
: A pointer to a buffer to store the result.

Remarks

The `CConsole::CONTEXTMODE` is defined as an enumeration as follows:

Context	Means
`CONTEXTMODE_GENERAL`	General registers.
`CONTEXTMODE_FPU`	Floating point registers.
`CONTEXTMODE_UBC`	User Break Controller.
`CONTEXTMODE_HBC`	Reserved - EVA chip’s Hardware Break Controller.
`CONTEXTMODE_ASE`	Reserved - ASE mode specific registers.
`CONTEXTMODE_PERF`	Reserved for Performance counters.
`CONTEXTMODE_TRACE`	Reserved for Execution Trace information.
`CONTEXTMODE_MMU`	Reserved for the Memory management unit.

The structure definition for the general and floating point registers is defined in `GenConsole.h` as the structures `SDIALRegsSH4EVA` and `SDIALRegsSH4EVA_FPU`.

A pointer to one of these structures is passed as the destination buffer to the function and the structure size, `length`, is specified using `sizeof(structure)`.

NOTE: *The context is declared to be represented in the native endian of the processor. For the Hitachi SH4 this is usually little endian. The endianness can be checked using the* `CConsole::ReadConfig` *command.*

WriteContext

Writes a context to the target.

```
DIAL_EC WriteContext(     DIAL_ID device,
                          CConsole::CONTEXTMODE mode,
                          WORD length,
                          const void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
A valid device identifier.

mode
A member of the `CConsole::CONTEXTMODE` enumeration that specifies which set of internal registers to read.

length
The size of the buffer storing the data to write.

buffer
A pointer to a buffer storing the data to write.

Remarks

The `CConsole::CONTEXTMODE` is defined as an enumeration as follows:

Mode	Context
`CONTEXTMODE_GENERAL`	General registers.
`CONTEXTMODE_FPU`	Floating point registers.
`CONTEXTMODE_UBC`	User Break Controller.
`CONTEXTMODE_HBC`	Reserved. Hardware Break Controller.
`CONTEXTMODE_ASE`	Reserved. ASE mode specific registers.
`CONTEXTMODE_PERF`	Reserved. Performance counters
`CONTEXTMODE_TRACE`	Reserved. Execution Trace information
`CONTEXTMODE_MMU`	Reserved. Memory management unit

The structure definition for the general and floating point registers is defined in `GenConsole.h` as the structures `SDIALRegsSH4EVA` and `SDIALRegsSH4EVA_FPU`.

A pointer to one of these structures is passed as the source buffer to the function. The structure size is specified as the `length` using `sizeof(structure)`.

NOTE: *The context is declared to be represented in the native endian of the processor. For the SH7091-EVA this is usually little endian. The endianness can be checked using the* `CConsole::ReadConfig` *command.*

ReadConfig

Reads the current configuration of the debug stub on the target.

```
DIAL_EC ReadConfig(       DIAL_ID device,
                          READCONFIG& config);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

config
: A `CConsole::READCONFIG` structure to hold the result.

Remarks

The configuration details retrieved in `CConcole::READCONFIG` include the endian of the target CPU, the CPU family and type. The Hitachi SuperH family is identified as an enumeration `UPROCFAMILY_HI_SH` and the processor type within that family is defined to be one of the following:

- `UPROCTYPE_UNDEF` (Reserved)
- `UPROCTYPE_SH1`
- `UPROCTYPE_SH2`
- `UPROCTYPE_SH3`
- `UPROCTYPE_SH3E`
- `UPROCTYPE_SH4`
- `UPROCTYPE_SH4EVA`

ResetNoDebug

Resets the main board without loading a debug stub and limits the available `CDial::Console` commands to `Inquiry` and `ResetAndDebug`. All `CDial::DA` commands are still available.

```
DIAL_EC ResetNoDebug(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
  A valid device identifier.

Remarks

Use this function to test applications without interference from the debugging system. For example, as if the application was running on a production unit.

ResetAndDebug

Causes a full reset of the system and the 1K ASERAM stub to be loaded onto the target and run. Full debugging functionality is available on the target.

```
DIAL_EC ResetAndDebug(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
: A valid device identifier.

Remarks

The reset behavior is dependent on a DA configuration status bit, `OSorCPUflag`, defined in `CDA::CONFIGDATA`. A value of 0 means OS debug mode and value of 1 means CPU debug mode.

When the DA is configured for OS debug mode, the target will automatically exit the debug stub and resume execution after the reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RUNNING` after `CConsole::ResetAndDebug()` has been successfully issued.

When the DA is configured for CPU debug mode, the target will remain in the debug stub after the reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RESET` after `CConsole::ResetAndDebug()` has been successfully issued and execution of the target will not occur until either a `CConsole::Resume()` or a `CConsole::Execute()` command is issued.

MakeSafe

`DIAL_EC MakeSafe(DIAL_ID device);`

Allows the DA to set the target microprocessor to a safe state by setting various registers.

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device
A valid device identifier.

Remarks

For the SH7091-EVA the following operations are carried out as a result of a `MakeSafe` command being issued:

- Register VBR is set to the debug stub default handler.
- Register DBR is set to the debug stub default handler.
- Register SR is set to the value 0x0600000F0 (status register).
- Register R15 is set to the value 0x0D000000 (stack pointer).

The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_SAFE` after a `CConsole::MakeSafe()` command has been successfully issued.

DIAL error codes

All error codes are defined as enumerations in *error.h*. The higher order 16 bits contain the error category and the lower order 16 bits contain the actual error code.

Error categories

Error Code	Means
ERRCAT_NOERROR	No error.
ERRCAT_SALSA	General errors.
ERRCAT_SALSADACON	Additional errors relating to DA and target.
ERRCAT_DA	Errors from DA specific commands.
ERRCAT_CON	Errors from target specific commands.
ERRCAT_WINSOCK	Errors Winsock specific commands.
ERRCAT_UNKNOWN	Errors of unknown origin.

SALSA error codes

Error Code	Means
ERRSALSA_NOERROR	No error.
ERRSALSA_NOTINITED	Call made to uninitialized layer.
ERRSALSA_DEVICENOTMATCH	Failed to validate the device type.
ERRSALSA_DEVICENOTFOUND	Didn’t find requested device.
ERRSALSA_CHANLNOTALLOCD	Logical channel (LUN) not allocated.
ERRSALSA_NOCHNLAVAIL	No more channels (LUNs) available.
ERRSALSA_MEMALLOCFAIL	Memory allocation failed.
ERRSALSA_BADADAPTER	Failed to validate the DA.
ERRSALSA_BADDEVICEID	Failed to validate the device identifier.
ERRSALSA_TIMEOUT	Timed out on command.
ERRSALSA_FAILCREATESYNCOBJ	Failed to create the device’s synchronization object.
ERRSALSA_FAILCREATELOCKOBJ	Failed to create the device’s synchronization control object.
ERRSALSA_FAILLOCK	Timed out or bad attempt to lock a synchronization object.
ERRSALSA_FAILUNLOCK	Failed attempting to unlock a synchronization object.
ERRSALSA_BADVERSION	The version passed to *SALSA.Initialise* does not match the version used to build the library.
ERRSALSA_DEVICEINUSE	The specified device is already in use.
ERRSALSA_NOTIMPLEMENTED	The specified function has not been implemented.

SALSADACON error codes

Error Code	Means
ERRSALSADACON_CMDPENDING	Command still pending completion.
ERRSALSADACON_CMDNOMATCH	Returned command header did not match in command field.
ERRSALSADACON_SEQNOMATCH	Returned command header did not match in sequence field.
ERRSALSADACON_FLSHOPENFAIL	Failed to open flash image file.
ERRSALSADACON_FLSHSTATFAIL	Failed to *fstat()* flash image file.
ERRSALSADACON_FLSHSHORTREAD	Short read from flash image file.
ERRSALSADACON_BADELEMENTSIZE	Bad *ELEMENTSIZE* specified in *ReadMemory()* or *WriteMemory()*.
ERRSALSADACON_NOCONPOWER	The target is currently powered off.
ERRSALSADACON_RESETINGCON	The target is currently being reset (Reset pin low).
ERRSALSADACON_FAILLOADSTUB (SIGNAL_FAIL)	One of the debug stubs failed to load.
ERRSALSADACON_FAIL1KSTUB	The ASE RAM debug stub failed to load.
ERRSALSADACON_FAILFULLSTUB	The Extended debug stub failed to load.
ERRSALSADACON_FAILNODEBUG (SIGNAL_NODEBUG)	No debug support available.

DA error codes

Error Code	Means
DAERRDA_DAAOK	Command completed without an error.
DAERRDA_DATMO DA	Command timed out during action.
DAERRDA_DAERA DA	Command error while erasing firmware.
DAERRDA_DAPRG	DA command error during reflashing.
DAERRDA_DAFIC	DA Command firmware image corrupt.
DAERRDA_DAERR	DA Command general error.

Console error codes

Error Code	Means
DAERRCON_CONAOK	Console command completed without an error.
DAERRCON_CONERR	Console command fatal error.
DAERRCON_CONBAD	Console command unknown.
DAERRCON_CONPRM	Console command parameter error.
DAERRCON_CONADR	Console command bad address.
DAERRCON_CONCNT	Console command bad count.
DAERRCON_CONCBF	Console command channel buffer full.
DAERRCON_CONCBE	Console command channel buffer empty.
DAERRCON_CONBSY	Console command not processed as BUSY.
DAERRCON_CONCNA	Console command not available (no debug stub).

WINSOCK error codes

Error Code	Means
ERRWINSOCK_INVALIDVERSION	The version of the Winsock library is invalid.
ERRWINSOCK_CANTCONNECT	Unable to connect to the specified device.

UNKNOWN error codes

Error Code	Means
ERRUNKNOWN_UNKNOWNERR	Error of unknown origin.